BRYAN SKINNER

GETTING THE MOST FROM YOUR ORIC

THE INDISPENSABLE GUIDE TO YOUR HOME COMPUTER

THE PENGUIN PERSONAL COMPUTER COLLECTION

Penguin Books

Getting the Most from Your Oric

Bryan Skinner was born in Worcester in 1954. He was educated at Manchester Grammar School and the University of Sussex, where he read Experimental Psychology, graduating with honours in 1976. After taking a Postgraduate Certificate in Education in Cheltenham he taught a variety of subjects including French, Biology, Psychology and Computer Studies to O and A level in a large comprehensive school in East London for several years. His interest in microcomputers led to several freelance programming contracts. One of these, a language translation by microcomputer project, resulted in his taking up the post of software support adviser with a software distributor in London. This interest also led to his involvement with the Natural Languages Translation Specialist Group of the British Computer Society. He has contributed regularly to *Popular Computing Weekly* and *Personal Computer News* and is currently a full-time journalist.

The advisory editors for the Penguin *Getting the Most from Your . . .* computer series are Rory Johnston and Martin Banks.

Rory Johnston is a freelance writer on science and technology, and a former Associate Editor of *Computer Weekly*. He worked previously in the computer industry and research, and as a teacher was a pioneer of computer education in schools.

Martin Banks was the first specialist columnist on personal computing in the United Kingdom. He now writes on the subject in a wide range of magazines including *Personal Computer World*, *Which Micro?*, *Computer Weekly* and *The Times*.

Bryan Skinner

Getting the Most from Your Oric

Penguin Books

Penguin Books Ltd, Harmondsworth, Middlesex, England
Viking Penguin Inc.,, 40 West 23rd Street, New York, New York 10010, U.S.A.
Penguin Books Australia Ltd, Ringwood, Victoria, Australia
Penguin Books Canada Ltd, 2801 John Street, Markham, Ontario, Canada L3R 1B4
Penguin Books (N.Z.) Ltd, 182–190 Wairau Road, Auckland 10, New Zealand

First published 1984

Copyright © Bryan Skinner, 1984
All rights reserved

Made and printed in Great Britain by
Richard Clay (The Chaucer Press) Ltd, Bungay, Suffolk
Set in 9/11 pt Monophoto Univers Light

Except in the United States of America, this book is sold subject
to the condition that it shall not, by way of trade or otherwise, be lent,
re-sold, hired out, or otherwise circulated without the
publisher's prior consent in any form of binding or cover other than
that in which it is published and without a similar condition
including this condition being imposed on the subsequent purchaser

Contents

Introduction: What is Your Home Computer For?

Most of us have a reasonably clear idea of what computers are used for in the world of business and industry: printing gas bills, forecasting the economy, controlling steel mills, operating robots on car assembly lines. But what is a home computer for? What can it do for you? There is a lot of uncertainty about that, and it is important to get it clear.

Small home computers work on exactly the same principles as big commercial machines but their capacity for handling information is very much less. Thus if you are thinking of opening a bank in competition with Lloyds or Barclays, your Oric won't do – you would need something much bigger. If on the other hand you operate a small business from home the Oric could be very useful, handling invoices, stock control, VAT and so on. A home computer like an Oric can deal with household accounts or work out mortgage repayments for people not in business, but most of us don't have that much accounting to do. Some of the jobs people suggest for computers in the home, like storing recipes, can in fact be done much better *without* a computer – in a good old fashioned book! With extra attachments the Oric can be used as a *word processor*, an electronic typewriter that makes it easy to produce neat, error-free typing, or as a *terminal* for obtaining information down the telephone line from such public data services as Prestel. But fundamentally a home computer is for two things: enjoyment, and learning about computers. For these two purposes, the Oric is excellent: quite apart from the enormously wide range of games that such a versatile machine provides, programming itself is a fascinating and stimulating pastime. Computers are becoming an important part of the world today and the only way it is possible really to understand and appreciate them is to use one.

The satisfaction you will get when the first program you have designed actually works is unlike any other experience in life. You will learn the reality behind all the mystery we see put about concerning this amazing new technology. You will find out (perhaps with some disappointment) the nonsense behind all the magazine articles that say 'Soon you will be able to buy a home computer that will wash the windows and mow the lawn for you', or that computers are going to take over the world. You will discover the true strengths and weaknesses of modern technology: what it can and cannot do, what it offers mankind and how it is fallible, and in the process perhaps you will gain some insight into the real value of human beings!

We are sure you are keen to get on with unpacking your Oric, so go ahead right now, and proceed to Chapter 1.

1. Getting Started

Unpacking

The Oric Atmos is essentially the same as the Oric-1. Where major differences exist, they will be explained in detail.

In the Oric's or Atmos's box you should find the following items:

1. The computer.
2. Mains power adapter with long lead attached.
3. Television lead.
4. Cassette lead.
5. Instruction manual.
6. Forth language cassette (48K models only)

The first thing you should do is to check for any damage. Make sure that the casing of the computer isn't cracked, that the four rubber pads on the bottom are all in place and that the warranty strip is firmly attached to the bottom. Check that the insulation on all the leads is sound and that the plugs at each end are properly attached. If you find any problems with any of these, you should return the entire package as soon as possible to your supplier. You should also do this if anything is missing.

Connecting up

The next thing to do is to connect everything together to make sure that it all works properly. To do this you should follow these steps in order:

1. Connect the computer to the television. The TV lead goes into the aerial socket at the back of your TV and into the right-hand socket at the back of the Oric. All references here to the rear sockets on the Oric are when looking at the computer from the front.

2. Switch on the TV. Any ordinary TV set will do, but a colour TV is preferable since one of the best features of home computers is their ability to use colour graphics. The TV is what the Oric uses to show you what's going on.

3. Plug the Oric's mains transformer into a mains socket. Don't worry about touching the plug at the end of the long lead attached to this – it only produces 9 volts and you won't feel this.

4. Plug the mains lead into the left-hand socket at the back of the Oric. This is right at the side of the casing, next to the long socket.

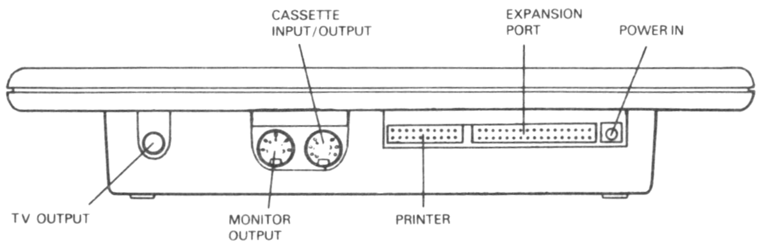


Fig. 1.1 Oric I/O (Input/Output)

Everything is now set up and you can start to tune the TV into the Oric's signal. This is usually somewhere after Channel 4. You may find that this takes some time and skill. When the TV is tuned in properly, you should get a message like this:

```
ORIC EXTENDED BASIC V1.Ø
© 1983 TANGERINE
4787Ø BYTES FREE
READY
```

If you have an Oric or an Atmos with version 1.1 of the ROM, you should see a slightly different message:

```
ORIC EXTENDED BASIC V1.1
© 1983 TANGERINE
xxxxx BYTES FREE
READY
```

If you cannot get this on your TV screen you may have a faulty computer or your TV may need some adjustment. Consult your computer dealer for more information. The words should be in black on white background. If you have a 16K Oric, the number of BYTES FREE will be 15102, instead of 47870. These figures refer to the amount of memory free for use by your programs. A 48K Oric can use a much larger (and hence more complex) program than a 16K version. Some programs for the 48K Oric cannot be used on the 16K machine because there simply isn't enough space available.

The keyboard

Look at the keyboard. It is through this that you control the computer, and all the computer's responses appear on the television screen. You will notice that the keyboard is very similar to that of a typewriter, with the same layout of the letters: QWERTYUIOP and so on.

Everything you type appears on the screen. The flashing square, called the 'cursor', shows you where on the screen the typing will appear. The computer is waiting for your instructions.

Everything that you say to the computer is a command, an instruction: 'Do this, do that . . .' Try typing this:

```
PRINT 3+5
```

This is an instruction to the machine to display something on the screen (the word PRINT is left over from the days when computers used teleprinters instead of television screens). When you have finished typing 'PRINT 3 + 5', you run up against your first 'computeristic' problem. The machine has no way of knowing whether you have finished that instruction, or whether you are going to type more. A human being could tell from your tone of voice, but there is no tone of voice in your typing! So, you have to tell the machine you are finished by pressing the key labelled RETURN (Carriage Return on a typewriter). Every instruction you ever give the machine will similarly have to be finished off with RETURN. Press it now.

What happens? On the screen now should be:

```
ORIC EXTENDED BASIC V1.Ø
 1983 TANGERINE

47870 BYTES FREE

READY
PRINT 3+5
8

READY
```

See the 8? The machine worked out the sum you gave it (3 + 5) and PRINTed the answer. Note that you could not just say 3 + 5 as on a calculator because the machine has to have a *command* to tell it which of its many functions you want. Try another sum:

```
PRINT 17+8-9
```

Don't forget to press RETURN at the end. Did the machine get the right answer? Didn't take long, did it? Try a multiplication. The 'multiply' sign is * to avoid confusion with the letter X.

```
PRINT 7*9
```

The word PRINT is one of a few dozen words that the Oric understands. These words make up its 'language'; there are many computer languages, and the one that the Oric uses is the best known – called Basic. If you give the Oric another word that is not in the Basic vocabulary, no matter how simple the word is, the machine will just not understand. Try typing:

```
HELLO [RETURN]
```

The machine replies:

```
?SYNTAX ERROR
```

It assumes you have made some mistake in your use of the Basic language, and it just tells you to try again. The same thing will happen if, for example, by mistake you type 'PRIMT' instead of 'PRINT'. One of the nice things about computers is that you cannot harm the machine by making this sort of mistake, no matter what you type on the keyboard. (You can of course harm it by dropping it on the floor!)

Now look at another word that *is* in the Basic vocabulary. An essential part of any computer is a memory, which can hold information while it is being worked on. You can tell the machine to save a number in its memory, but to do that you must put a label on the number so that you will be able to call it back again, just as in putting a label on a suitcase you leave in the Left Luggage Office. With the computer the label is a letter, any letter you choose. The command word is LET. If you want to store the number 157 in the memory, the command looks like this:

```
LET A=157
```

This means, 'Store the number 157 in the memory and call it A.' Then if you say:

```
PRINT A
```

you get the response from the computer:

```
157
```

It remembered the number! It will continue to remember it until you do something such as change it, clear the memory or switch off the computer.

Now try typing this:

```
LET A=A+1
```

Then give the command

```
PRINT A
```

Do you see what has happened? You get the answer 158. The machine has taken A, added 1 to it, and made that the new value of A, destroying the old value. You will find this trick very useful later on.

A simple program

The machine can also hold *instructions* in its memory, and it is this ability that really gives computers their power. A complete set of instructions for a complicated task can be assembled in the memory and then carried out in sequence automatically, very fast. This set of instructions is called a *program* (always spelt without the *-me*). Try typing in a very simple program, consisting of these two lines:

```
1 PRINT "your name"; [RETURN]
2 GOTO 1
```

To get the quote marks " you have to hold down one of the SHIFT keys while pressing [,"]. This applies to many keys which have two different inscriptions on them: when pressed without the SHIFT key, the lower marking applies, but when pressed with SHIFT, the upper marking applies. Do not miss out the semicolon at the end of the first line.

If you hit a wrong key, don't worry. It is easy to delete a wrong character by pressing DEL. You will see that the character disappears without a trace. This is useful because in a computer instruction, every single character, every last dot and comma, has to be exactly right – otherwise the command will not work. While a human would easily be able to spot (and correct) an error of a single character, computers are not 'intelligent' enough to do that yet.

When you have finished typing these two lines, type 'RUN [RETURN]'. The screen should fill with your name, then you will notice that the screen flickers slightly at the bottom. This is because the program is still printing your name, but so fast that you can't see the changes. The program is continually adding your name at the foot of the screen, making all the other words move up a line and disappear off the top of the screen (this process is known as 'scrolling').

When you've had enough, you can press the key marked CTRL and, while pressing this key, the letter C. This action will stop your self-advertisement. CTRL stands for CONTROL and this key can be used in conjunction with other letters for different effects. Pressing CTRL and C will be referred to as CTRL C. CTRL L will clear the screen of all text and is useful for getting rid of unwanted text to make things clearer.

You have just run a computer program – an extremely simple one but a program none the less. Before, when you gave the computer instructions it carried them out and then forgot them – this time it has *remembered* the instructions. They are still there. Try typing 'RUN [RETURN]' again. The screen fills up with your name again, right? Press CTRL C. Why did the machine remember the instructions? Notice that in front of each instruction that you typed there is a *number*. This tells the computer in what order to carry out the instructions, and it also tells the machine, 'This is an instruction to be stored in the memory, to be carried out when the user types RUN.' Exactly how the instructions in this program work is explained shortly.

Loading a program from tape

Most computer programs are much longer and more complicated than that one. It takes a long time to type them in, so to save you having to do that, they can be stored on magnetic tape cassettes, exactly the same as those used with audio cassette players.

You can connect an ordinary portable cassette recorder/player to your Oric with the cassette lead supplied. The large grey plug goes into the third socket from the right at the back of the machine. The other end of the lead has two jack plugs. One of these goes into the socket on the cassette deck marked

EAR – this is for getting programs recorded on tape into the Oric. The other jack plug goes into the socket marked AUX or MIC – this is for recording programs in the Oric's memory on to tape.

If you bought a 48K Oric you will probably have been supplied with a cassette with some demonstration programs on it. (You may also have a cassette with the computer language Forth on it, but this is outside the scope of this book.) If you don't have a demonstration cassette, you should buy a program on tape so that you can learn how to LOAD a program from tape to the Oric's memory.

Getting a program from tape into the Oric's memory is fairly simple:

1. Type 'NEW' – this will clear the Oric's memory.
2. Type 'CLOAD""' – this tells the Oric to look for the first program it finds on the tape and load it into memory. As soon as you press RETURN, you should see the message SEARCHING . . . at the top left of the screen.
3. Press the PLAY button on the cassette deck. After a short while, the SEARCHING message should change to LOADING . . . If this doesn't happen you will have to rewind the tape and change the tone and volume settings of the tape player. The Oric needs both these to be set to a high value – plenty of treble and volume.

Once a program has been successfully LOADed from tape, it may automatically start RUNning itself. If it doesn't, you'll see the message 'LOADING . . .' disappear and the Oric will display the READY message again. At this point you should type 'RUN' to make the program start.

You will probably find that it takes quite a few trials to find the correct settings for tone and volume on your tape deck. You may even find that your tape deck will not play the program properly to the Oric so that you just can't use it to LOAD programs. More information on using cassettes is given later.

When you type 'CLOAD', you will notice that the cursor stops flashing and that the Oric will not respond to the keyboard at all while it's SEARCHING for a program. If you can't get a program to LOAD successfully, you'll either have to disconnect the Oric from the mains, then plug it back in (this is best done by removing the plug at the rear of the computer) or press the RESET button. The latter method is preferable, since the power sockets can be somewhat fragile on some machines. The RESET button is inside the underneath of the casing, at the left-hand side. You will need something quite thin to get at it and it only need be pressed once, gently. Pressing RESET causes the Oric to stop whatever it's doing and return control to you at the keyboard. It can be used to stop some programs as an alternative to pressing CTRL C.

Some programs have to be LOADed with the command 'CLOAD"",S'; more information on this is provided later.

While the Oric is loading a program it does some error checking. If it comes up against a problem it will stop the loading process and report the error with the message 'FILE ERROR – LOAD ABORTED'. You will have to rewind the tape to the beginning of the program and try loading it again. The most common reasons for the problem are too high a volume setting or a faulty recording. In the former case a lower volume setting will put things to rights; in the latter you may find it impossible to load the program at all.

Your first program

Now that you have seen some programs in action, you will want to find out how they are written, so that you can write some for yourself. To start with, let's look back at that very crude program you ran before:

```
1 PRINT "your name"; [RETURN]
2 GOTO 1
```

It consists of only two instructions. The first you are familiar with: PRINT, although in this case you want it to print not numbers but words. You can easily make it do that by enclosing the words in quote marks " ". Why do you need the quote marks? Try typing these lines:

```
LET A=21
PRINT A
PRINT "A"
```

Do you see the difference? The letter A could be just a letter you wanted printed, or it could be the label of a number you had stored in memory. The quotes tell the computer, 'These are real letters, not labels.'

Instruction number 2 in the program is new. Normally the computer carries out the instructions in its memory strictly in the order of the numbers. Occasionally you want to break the sequence. The instruction GOTO tells the machine to 'go to' the instruction number specified and carry on executing the program from there – in this case, line number 1. So it does 1 (for the second time) and comes to 'GOTO 1' again. This makes it jump back yet again, and so on in what is called a 'loop'. When will it stop? Never! Fortunately you have CTRL C to interrupt this process, or else it would go on until the end of time – the computer never gets bored and gives up.

Retype line 1, leaving out the semicolon at the end of the line. RUN the program again. You should see that if you want things to appear on the same line, you have to put a semicolon after each item. If you don't, each PRINTed item will appear on a new line.

Let's now try writing a slightly longer program that actually does something useful. We are all familiar with the idea of having a rule to work out how long to cook something. For chickens, say, the rule could be: Cook for twenty minutes for each pound it weighs, and then an extra twenty minutes. This could be written like this:

TIME=WEIGHT × 20+20

Suppose you have a meat factory which turns out pre-cooked chickens by the hundred. For each one, the time has to be worked out. The computer can do this for you very easily. It can take the weight of each chicken, calculate the time, and PRINT it for you.

To do this job, the first thing your program needs to do is find out the weight of the chicken. You have to tell it that by typing the weight on the keyboard, and the computer has to be told to wait while you type the number, then take it from the keyboard and store it in the memory. This requires a new command word INPUT, which is the first instruction in the program.

It is a good idea, rather than numbering your instructions 1, 2, 3 and so on, to use the numbers 10, 20, 30, etc., to make it easier to add further instructions

later on should it become necessary. So designate the first command in the program '10', so that it looks like this:

```
10 INPUT W
```

Notice that the figure zero has a line through it so that you can tell it apart from the capital letter O: this is important. This instruction means, 'Wait until a number is typed on the keyboard, and when that happens, store it away in the memory with the label W.' You can use any letter you like for the label but W is a good choice in this case because it is easy to remember that it stands for Weight.

When the computer has got the 'data', as it is called, it can proceed to calculate the answer. This is a simple piece of arithmetic, as in the first few examples of commands you tried giving to the machine. You can use the label T to store the time, so the second instruction of the program looks like this:

```
20 LET T=W*20+20
```

Do you see how that works out the time by the formula we decided on earlier? Now the computer just has to tell you the answer, and the command for that is:

```
30 PRINT T
```

The complete program looks like this:

```
10 INPUT W
20 LET T=W*20+20
30 PRINT T
```

Type NEW to clear out the memory, then type those three lines in and give the command RUN.

You should immediately get on the screen:

```
?
```

The computer prints a question mark to tell you it is waiting for you to type in some data (it has encountered the INPUT instruction). Suppose the chicken weighs three pounds, so you type

3 [RETURN]

In an instant the computer replies

```
80
```

Eighty minutes is the correct cooking time! Now try typing RUN again for another chicken, say one and a half pounds. In response to the ? you type

```
1.5
```

and you should get the answer

```
50
```

If you get any other answer, you have made a mistake in typing in your program. Remember that every single character has to be correct. Carefully compare what you typed with the three instructions as given above. You can make the machine show you the instructions it has in its memory by giving the command LIST. If an instruction is wrong simply type it over again, including the number at the beginning.

When your program is working properly, try running it a few more times with various weights of chicken. You have succeeded in writing a real computer program! It is very simple but it is no different in principle from the huge, complex programs that are making such a valuable contribution to our society. You will probably go on to write much more elaborate programs yourself, and also to use programs that other people have written.

The computer explains itself

To see how some more advanced ideas in programming work, try making some improvements to the chicken program. To start with, it would be nice to get the program to explain itself. At the moment you have to know that the ? means the computer is waiting for input: you can easily use the PRINT command with words in quotation marks to get the computer to say what is going on. It needs to give the explanation *before* it types the ? so this new command has to have a *lower* number than the INPUT instruction. Type in this:

```
5 PRINT "WEIGHT, PLEASE"
```

Now type RUN and see what happens. You should get

```
WEIGHT, PLEASE
?
```

Try using the program in its new improved form a few times. With explanations such as that one included in programs, you can see how computers can be used by people who have no idea how the programs work. They just follow the instructions (assuming that the instructions are adequate). Naturally, as well as explaining the question, the program ought to explain what the answer is. Before typing out the answer it should say: 'THE COOKING TIME IS'. See if you can work out for yourself what instruction is needed to make 'THE COOKING TIME IS' come out before the answer. (The correct instruction for this is given at the end of this chapter.)

It would be nice not to have to type RUN for every chicken. A further improvement you can make to your program saves you having to do this; the program can loop back to the beginning automatically every time it has finished a calculation. To do this simply add the instruction

```
4Ø GOTO 5
```

This will make the computer go back to instruction 5 instead of stopping after instruction 30. Try typing RUN now and see this work.

The only problem with this is, how do you get out of the loop and do something else when you have had enough of chickens? When the machine is waiting at the INPUT instruction with a ? on the screen you have to break out by CTRL C.

You have been fiddling around with this program quite a lot since you started; to see how it looks now in its improved form, type LIST. Notice that the lines come out in the order of the numbers, not the order you typed them in. The program is neat and complete in its final form – there is no trace of any mistakes you may have made and subsequently corrected.

Doing it yourself

By far the best way to learn is by doing, so you may like at this stage to try writing one or two easy programs of your own. These could be: finding the area of a circle, finding the area of a rectangle, or converting degrees centigrade into degrees Fahrenheit. The circle uses the formula we all learnt at school: $A = \pi r^2$, where r is the radius. In computer terms this would be: 'A=R*R*3.14' (π is 3.14, approximately). The area of a rectangle, of course, is base times height, so in that case it is necessary for the program to ask the human user for *two* items of data, with two INPUT statements.

Centigrade into Fahrenheit uses the formula:

$$F=C\times 1.8+32$$

or in words, multiply the number of centigrade degrees by 1.8 and add 32.

The opposite, converting Fahrenheit into centigrade, is slightly more complicated. The formula is

$$C = \frac{(F - 32)}{1.8}$$

or, first subtract 32 from the number of Fahrenheit degrees, then divide by 1.8. The divide sign in Basic is / as in PRINT 12/4, and you can either do the arithmetic in two steps or use brackets ( ).

Remember to type NEW to clear away the chicken program or whatever else is in the memory before you type in your new program.

As a final example in this introductory chapter, try this program, which shows off some of the Oric's facilities for handling colour. There are several instructions here which have not yet been explained. Just type them in as they are; they will be explained in detail in later chapters.

```
1Ø CLS
2Ø FOR L=1 TO 1ØØ
3Ø PRINT "your name";
4Ø NEXT
5Ø FOR K=1 TO 7
6Ø PAPER K
7Ø WAIT 5Ø
8Ø NEXT
9Ø GOTO 5Ø
```

If you wanted to have your name printed once only on each line, you would miss out the semicolon (;) at the end of line 30. The semicolon serves to tell the Oric not to move the cursor to the beginning of the next line after displaying something on the screen.

Additional explanation line for chickens program:

```
25 PRINT "THE COOKING TIME IS"
```

2. Telling the Oric What to Do – First Steps in Basic

The Oric's keyboard looks very like that of a typewriter, but it behaves in some different ways. So far all letters you have typed have appeared on the screen in upper case (capitals). Press CTRL T, then type some letters. As you will see you're now in 'lower-case' mode. If you try to use some of the Oric's special words like RUN or LIST in lower case you'll get a '?SYNTAX ERROR' message, because the Oric can only understand words in upper-case letters. In lower-case mode you can use the SHIFT key to make the letters come out in capitals, just like a typewriter. Press CTRL T again. You should see the message CAPS at the top right of the screen. For some odd reason, when you turn on the Oric, although it's in upper-case mode, it won't tell you this until you go into lower case and then back into upper case.

You can move the cursor around the screen by pressing the arrow keys. You've probably found out by now that if you keep a key depressed it will 'repeat' after a short while. This is very useful for moving the cursor when you want to change program lines.

Enter LIST after breaking out of the previous program (using CTRL C). Now press the up-arrow key until the cursor is opposite line 70 of the program. Now press CTRL A until the cursor is on the 0 after the 5. Now press RETURN, then CTRL L, then type LIST again (CTRL L clears the screen of all text). You should see that line 70 now reads:

```
70 WAIT 5
```

RUN the program again. The colour changes are now so fast that you can't see all the different colours clearly. Stop the program (with CTRL C) and, using the method outlined above, get the cursor to the space after the 5 in line 70. Now type two zeros and press RETURN, then CTRL L again. If you LIST the program again you will see that line 70 now reads:

```
70 WAIT 500
```

If you RUN the program now you will see how you can use the WAIT command in a program to make things pause for a long or short time.

CTRL A copies characters under the cursor into the computer; these characters can then be stored or acted on when you press RETURN. Using CTRL A can save a lot of typing as it means that you don't have to constantly type in new versions of program lines; all you have to do is to copy an existing line to the point you want to alter, type the new characters, then press ENTER.

The CTRL key can be used with several of the other keys for different effects. CTRL F will turn off the keyclick and pressing CTRL F again will turn it back on. CTRL Q will turn off the cursor, repeated it will turn it back on.

CTRL S turns the screen off so that nothing is displayed, again, repeating the command will turn the screen back on. These actions which alternately turn something on or off are called 'toggles' and are discussed in more detail in later sections.

As you can see from the keyboard, the Oric has the mathematical operators + and −. In Basic, the operators multiply and divide are shown as * (asterisk or star) and / (oblique). If you enter 'PRINT 3/2' the Oric will display the answer: 1.5. All mathematics are done with decimal numbers, not fractions. You can use these operators in programs, rather as you would using a calculator.

One thing you *do* have to be careful with, however, is the order of the operations you ask the Oric to do. Try entering 'PRINT 2+1*3'. The answer the Oric gives is 5, not 9 as you might have expected. This is because the Oric (and indeed all versions of Basic) perform mathematical operations in a command in a certain sequence; *, /, +, and −. This is known as the order of precedence. If you want the Oric to tell you the result of adding 2 to 1, then multiplying that result by 3, you have to put brackets (parentheses) around the part of the expression you want done first: 'PRINT (2 + 1) * 3'. Brackets have a higher precedence than *, so the expressions inside them are always done first. If you have a number of brackets in a line, the Oric works from left to right and this is also true when there are a set of equal precedence operators in a line.

You also have access to what are known as relational operators: =, > and <. These symbols represent 'equals', 'greater than' and 'less than'. You can combine the latter two with the former to get 'greater than or equal to' (>=) and 'less than or equal to' (<=). <> means 'not equal to'.

These operators are most often used in programs to decide what to do if the value of some variable falls below, or rises above, a set value. You can also use them to compare the values of two (or more) variables.

For example, if you're writing a game in which the player has a number of lives, you will want the game to end when the player has lost all the lives available. You might keep track of the number of lives in the variable L. Each time the player lost a life you would subtract 1 from the value in this variable: 'LET L = L −1'. You would also test for L = 0 like this:

```
IF L=Ø THEN CLS:PRINT "GAME OVER":END
```

Notice how you can use the colon to put more than one statement or command on a single program line. This saves both space and time, and is particularly useful after IF . . . THEN clauses.

The Oric also gives you some operators which are words, not symbols. These are AND, OR and NOT. They are known as Boolean operators (after George Boole) and will be discussed in later chapters.

If you try to type in a program line which is more than eighty characters long, when you reach the eightieth character, the Oric will produce a ping sound and display an oblique (/) at the point where the line exceeded the maximum length. You will have to split the line into smaller segments over two or three lines if this happens.

If a line is too long to fit across the screen, the Oric will wrap it round on to the next line. You'll notice that program lines begin two spaces from the left-

hand margin and if a line carries over on to another line, that there appears to be a gap in between the end of one line and its continuation in the next. This has no adverse consequences, but is a result of the Oric's use of the first two columns of each row to hold the INK and PAPER attributes (colours) for that row or line.

When you are entering a program you are bound to make mistakes. Fortunately you do not have to retype a whole line to correct a single letter. The Oric has a number of functions which allow you to change the contents of a program line quite simply.

When you are LISTing a long program, pressing the space bar will suspend the LISTing until you press it again. If you want to stop the LISTing, press CTRL C. Unfortunately, a bug in the ROM (version 1.0) means that after pressing space, then space again (to restart the listing), you may have to press CTRL C a few times before the LISTing stops.

Having found the line that you wish to change, you can move the cursor about the screen with the arrow keys to the beginning of the line you want to change. Then, using CTRL A, you can copy the line into the keyboard buffer, making changes as you go. You can, while doing this, use the up or down arrows to insert text, then reverse this to get back to the position from which you left. Pressing RETURN will dump the contents of the buffer into the Oric which will then enter the amended line into your program. If you get into a mess, CTRL X will erase the changes you have made, so you can start again. If you know which line you want to change, you can enter 'EDIT *line number*', which will produce the offending line for editing. LIST can be used as in 'LIST-100', 'LIST 100–200' and so on. 'LIST-100' will display all program lines (if any) up to and including line 100; 'LIST 100–200' will list all lines from 100 to 200 inclusive.

The programs you wrote in Chapter 1 were very simple. In fact, for the chicken-cooking program you did not really need a computer – you could have done the arithmetic in your head just as easily. For a computer to be useful, it has to have a much more complicated task, involving many steps and countless repetitions so that the speed and reliability of the machine can come into their own. The snag here is that the computer cannot work out for itself what those steps have to be. A human being (the programmer) has to do that, and the job can be long and complicated (but interesting and rewarding as well). Of course, when it is done and the program is written, the human can relax and let the computer do all the work, but prior to that the programmer has a lot of thinking to do.

The programmer has to break down whatever task is at hand into a series of steps, each of which is small enough for the computer to handle. The steps are then written out in whatever computer language is being used; the particular language determines how small the steps are. The programmer has to make sure to get all the steps clear and in the right order.

As an example of this process of breaking down a job into steps (one totally unconnected with computing) consider the task of changing a wheel on a car. The steps might be:

Apply handbrake
Get jack and brace

Remove hubcap
Loosen wheelnuts
Jack up car
Remove wheelnuts
Remove wheel
Get spare
Mount spare
Replace nuts
Lower car
Tighten nuts
Replace hubcap
Replace jack, brace and punctured wheel in car

There is a well-known trap of course: you must remember to loosen the wheel-nuts *before* jacking up the car, or else the wheel will spin round. Try the same exercise for yourself, with 'Filling a fountain pen' or 'Making a pot of tea'. When exactly does the pot get warmed? Possible answers to these two tasks are given at the end of the chapter.

While these are amusing examples, there is one crucial element that they are lacking. None of them involves decisions or questions, while in real life these arise all the time. Any process that involves decisions is best shown in the form of a *flowchart*, as shown in Fig. 2.1. Here the ordinary instructions are shown in rectangles, but occasionally there are questions that have to be asked. These are held in diamonds, and according to the result of the question, the flowchart branches in one direction or another. Having looked at the example, try drawing a flowchart yourself for 'How to cross the road' or 'Making a telephone call from a coin box'. For the first, the questions follow the lines of 'Look right. Is there anything coming? Look left . . .' The problem is that if you wait till the right is clear, then wait till the left is clear, some traffic might have reappeared on the right in the meantime; but you must make sure that the flowchart does not stick you in a loop so that you *never* get across the road. With the telephone example, the questions would include: 'Is there a dial tone? Is there an engaged tone? . . .'

Answering questions in computer programs and making decisions on the results requires a new command word, which is IF. This works by testing to see whether some condition is true or not. If it is true, the program may carry out a GOTO to some line which you specify; if it is not true, the program carries on in the normal sequence. It can test whether one number is equal to another, or greater than or less than another. There are other things that can be tested for but basically every question in a program has to be reduced to the form 'Is one number equal to, or greater than or less than, another?'

A typical IF instruction looks like this:

```
5Ø IF A=7 THEN GOTO 11Ø
```

If the number labelled A in the memory is equal to 7, the program will immediately go to line 110 and carry on executing from there; if A is anything else, the next line following 50 will be executed. Here is an example of a very simple program using IF. It decides which of two numbers the user types in is bigger.

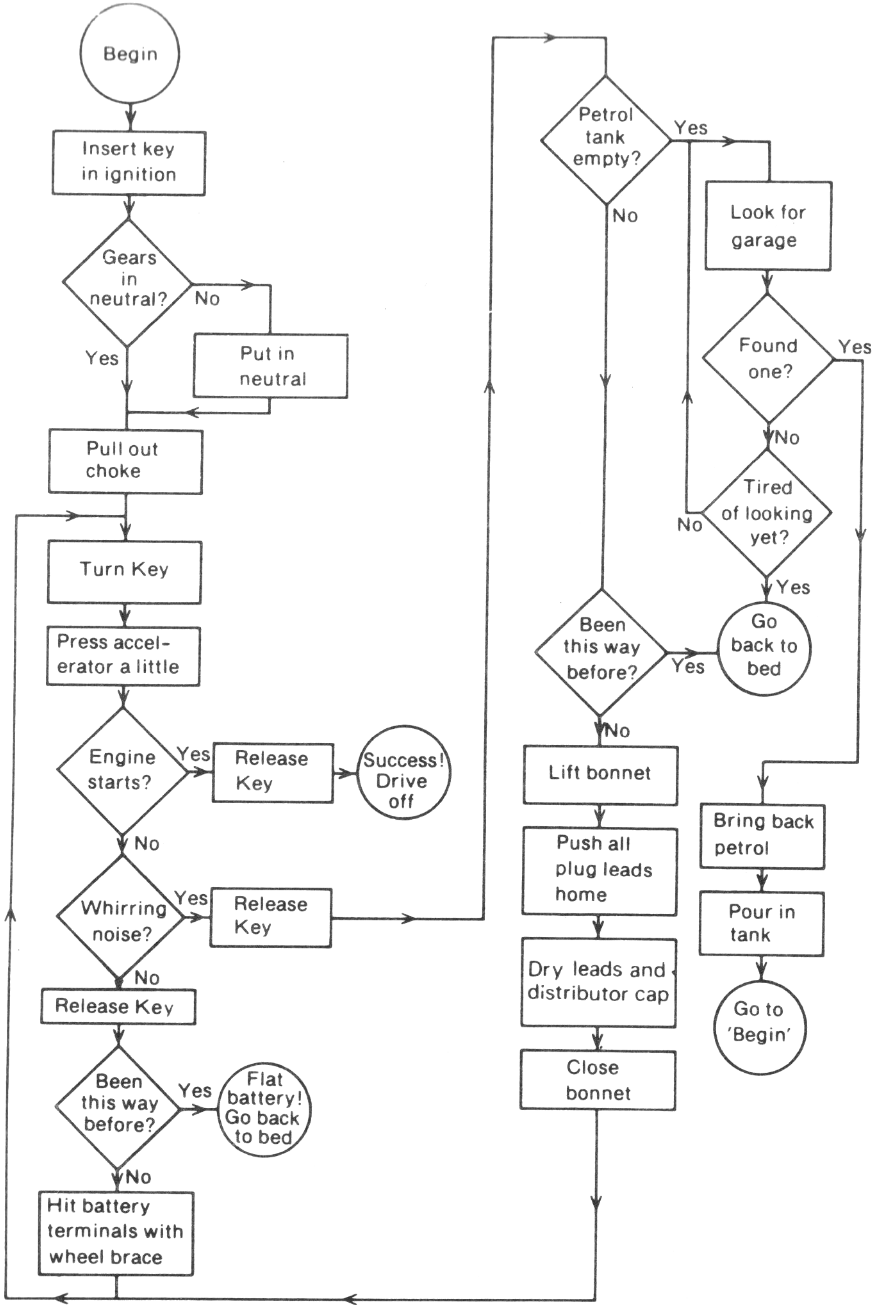


Fig. 2.1 Flowchart for 'Starting a car'

```
 1Ø PRINT "TELL ME A"
 2Ø INPUT A
 3Ø PRINT "TELL ME B"
 4Ø INPUT B
 5Ø IF A<B THEN GOTO 8Ø
 6Ø IF A>B THEN GOTO 1ØØ
 7Ø PRINT "A=B":END
 8Ø PRINT "A IS SMALLER"
 9Ø END
1ØØ PRINT "A IS BIGGER"
```

There is another new command word at line 70: END. This simply stops the program. Can you see what would happen if it were not there?

An average task

To see how flowcharts are used in programming, let us now write a program that actually does a useful job, namely, working out the average (or 'mean') of a set of numbers. This is done, of course, by adding all the numbers together and dividing by however many numbers there are. A program to do this cannot simply say 'Add the first two', 'Add the third', 'Add the fourth' . . . because it is not known beforehand how many numbers there are – one time there could be nine numbers, the next time twenty-three, the next time a hundred, and so on. Consequently the program must go round in a loop, adding each number to an accumulating total, until all the numbers have been received. To do this it must first find out from the user how many numbers there are. Then it needs to *count* the numbers as they are fed in, and stop after the last one. Then it should do the division and print the answer. This whole process is shown as a flowchart in Fig. 2.2.

One location in the memory is used to hold the running total, and another to keep the count. Each time round the loop, 1 is added to the count, and then the question is asked, 'Has the count reached the required tally yet?' In other words, 'Have we got all the numbers yet?' If No, the program loops back and gets another number. If Yes, it does the division and displays the answer. The running total and the count both have to be set to zero at the beginning – this is a commonplace requirement in programming that is called *initialization.* All of these functions can be seen in the flowchart.

In Basic, the obvious thing to do is to use the label S to store the running sum, and C for the count. Each individual number is read in with the INPUT statement

```
INPUT X
```

and then it is added to the sum by the same trick we saw in Chapter 1:

```
LET S=S+X
```

Adding 1 to the count is done by

```
LET C=C+1
```

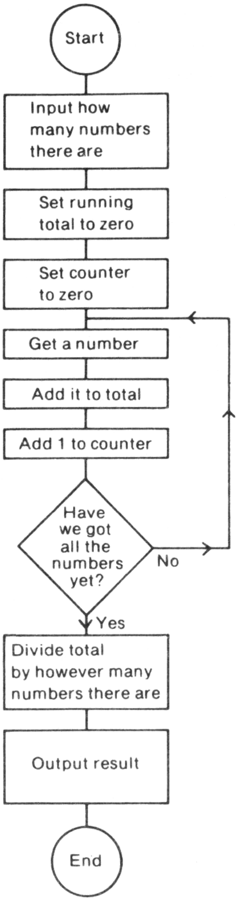


Fig. 2.2 Flowchart for finding averages

Thus the whole program in Basic looks like this. The explanations to the right of each line are not part of the program.

```
10 INPUT N                  Get how many numbers
                            there are.
20 LET S=0                  Clear running sum.
30 LET C=0                  Clear count.
40 INPUT X                  Get a number.
50 LET S=S+X                Add it to sum.
60 LET C=C+1                Add 1 to count.
70 IF C<N THEN GOTO 40      Got all the numbers
                            yet?
80 PRINT S/N                Yes – do division to
                            find average.
```

Note how closely the program corresponds to the flowchart. You might like to try typing in this program and seeing how it works. An obvious improvement you could make would be to provide PRINT statements to explain the requests for data and the output.

At this point you could write a program of your own on similar lines. This would make your Oric act as a cash register, adding up numbers as they are typed and printing out the total. One of the good things about a general-purpose computer is that it can do the work of a large number of different specialist machines – a computer can be a cash register but a cash register can only be a cash register and nothing else. The only problem you have is that on a cash register there is a special button marked 'Total', to tell the machine when you have finished feeding in amounts and you want the answer. There is no such button on the Oric, so how do you tell it to print the total? It would be a nuisance to have to do it the way the averaging program worked, by you counting all the items beforehand and telling the program how many there are. A better trick is to have a special code that tells the computer, 'That is the end – now print the total.' When one particular number is typed in when the machine is asking for data, it tells the program, 'This is not a price of an item to be added, but a signal to tell you to print the total.' An obvious choice for this special number would be 0, as no item has a price of £0.00, so there could be no confusion between a price and the 'End' code.

Try drawing a flowchart for the cash register program, and then writing it in Basic. Immediately after the INPUT instruction you should have an IF command which tests to see whether the number just obtained is 0 or not. If it is not, it is added to the accumulating total and the program branches back with a GOTO to get another number. If it is equal to 0, the program branches to the end and prints the total. Basically the program is very similar to the averaging one.

Saving programs on tape

As you will be aware, when you turn off your Oric any program that is in the memory is lost. You may write a long program and want to use it again another day and it would be a nuisance to have to type it all in again, making mistakes along the way. To avoid this, you can 'save' (record) programs on to tape, using a cassette unit. As it is possible to record many programs on to one tape, you can give each program any name you like so that you can later tell the Oric to LOAD that program.

Suppose that you are going to save a program, calling it CHARLIE. Put a blank tape into the cassette recorder (remembering that you can record on either 'side' of the tape) and rewind it to the beginning. Now press PLAY on the recorder until you are sure that the leader tape (the transparent section at the beginning of the tape) has passed the record head. This usually takes about five seconds. Stop the tape. Type 'CSAVE "CHARLIE" '. *Before* you press RETURN on the Oric's keyboard, put the recorder into record mode by pressing RECORD and PLAY. Now press RETURN. After a few moments you should see the message 'SAVING ... CHARLIE' at the top left of the screen. You will also notice that the cursor has stopped flashing and that the

Oric will not respond to the keyboard at all. When the Oric has finished saving your program, it will start flashing the cursor again and display the READY message. You can now turn off the tape recorder.

It is always possible to overwrite programs on tape with newer versions or with other programs, but you need to be careful, as this can lead to complications. Suppose that on a tape there are these programs:

CHARLIE	LUCY	SCHROEDER	LINUS

You might want to overwrite SCHROEDER with SNOOPY. To do so, you would first position the tape at the end of LUCY. This will work, but if SNOOPY is longer than SCHROEDER, you will overwrite the beginning of LINUS. If SNOOPY is shorter than SCHROEDER, the bit of SCHROEDER left sticking out at the end of SNOOPY will not matter. In fact, it's not a good idea to overwrite tapes in this way. It is better first to 'wipe' the tape using a good hi-fi system to ensure that the tape is 'quiet', i.e. that all recordings have been erased.

Many computers have a VERIFY function which the Oric (version 1.0) lacks. This function makes the machine check that everything recorded on a tape is exactly the same as the program in memory. This allows you to check that a program has been recorded successfully before you switch off the computer or load another program. Since you can't do this with the Oric (version 1.0), it's a good idea to record every program twice as a safeguard.

The Oric has two program-saving functions that most micros don't have. One of these is the ability to CSAVE programs at two different speeds, fast or slow. This is particularly useful if you have a large program that would take a long time to LOAD using the slow speed. The disadvantage is that SAVEing programs at fast speed may cause LOADing errors if your tape machine is not of a high enough quality. That is, because the data is being sent at a faster rate using the fast speed, your tape recorder may not be able to reproduce the information sufficiently well for the Oric to recognize all the data.

Reliability

When you save a program using 'CSAVE "LUCY" ', the Oric 'defaults' to a high-speed save. If you want to save a program at the slower speed (in order to be more sure of a successful SAVE) you have to add ',S' to the command – 'CSAVE "LUCY",S' does this. It is important to remember how you saved a program. If you CSAVEd a program using ,S for a slow CSAVE, then you must CLOAD it in the same way, i.e. 'CLOAD "LUCY",S'. If you try to CLOAD a program that was CSAVED with ,S and forget to add ,S after the CLOAD, the program will not CLOAD.

Auto-run

The other very useful facility in this area offered by your Oric is that you can save programs in such a way that they will RUN themselves as soon as they have been CLOADed. To do this, just add ,AUTO after the CSAVE statement. For example, suppose that you wanted SNOOPY to RUN the instant it was CLOADed, you would type 'CSAVE "SNOOPY",AUTO'.

You can even mix the slow CSAVE and AUTO-run commands together, so 'CSAVE "SNOOPY",S,AUTO' will CSAVE SNOOPY on tape at a slow speed and when you type 'CLOAD "SNOOPY",S' the program will CLOAD, then RUN automatically. Notice that you don't have to specify that the program has been CSAVEd as an AUTO-run program: the Oric will work that out for itself. For those interested in such matters, the Oric usually CSAVEs at 2,400 baud: the ,S option saves at the painstakingly slow rate of 300. If you enter 'CLOAD""', the Oric will load the first program it comes across on the tape.

Unfortunately, the Oric will not tell you the names of programs on tape if you use CLOAD"". This can be a real nuisance if you've forgotten exactly where you've put programs, or if you get your unlabelled tapes mixed up. Some machines will 'look' at the tape and tell you the name of the program that's being loaded. As the Oric can't do this unaided, you would have to CLOAD each program in turn, then RUN or LIST it until you found the one you were looking for.

However, the following little program may save you hours of frustration. If you type this in, then RUN it, then CLOAD "" and then CALL 1024, lo and behold, the name of the program that's been loaded will be displayed.

```
10 REM program loading name-revealer
20 FOR D=1024 TO 1031: READ A$
25 V=VAL("#"+A$)
30 POKE D,V
40 NEXT
50 DATA A9,49,A0,00,20,ED,CB,60
```

One very unusual feature of the Oric's tape-handling system is that your program names may be up to seventeen characters in length. Most other machines restrict you to eight characters, which is not really enough. Seventeen characters allows you to include the date at the end, so that you know which program was recorded when. This is useful if you're developing a long program over several months as it's very easy to forget which version is the latest. It is therefore always a good idea to also put the date (and perhaps time) the program was CSAVEd in a REM (REM stands for 'remark' – see p. 156) statement in the first few lines of a program, for example:

```
10 REM 9.30 am 05/03/'84
```

Tips

Tape recorders are not the most reliable means of data storage and you should bear the following points in mind.

Before you buy your micro, get your cassette deck serviced. This will make sure that the recording head is at the correct angle to the tape and is clean. One of the most common problems caused by cassettes is that this 'azimuth' gets out of alignment. You should also clean the recording head fairly regularly. Don't use a tape-head cleaning cassette as these don't usually work very well. Use a short stick with its tip bound in cotton wool soaked in meths or other solvent. Put the cassette in PLAY mode and wipe the head which protrudes when PLAY is engaged. Clean the pinch roller as well as this can collect dirt and oxide deposits.

Don't ever use the first few seconds of tape after the 'lead-in' section of a cassette. It's always good practice to watch for the leader to clear the recording head, then count to five slowly before recording a program. The reason for this is that when a tape is rewound, the first few inches can be stretched slightly when the tape reaches the beginning. This is also true if you 'fast forward' a tape.

Don't use C120 tapes. These can strain the mechanism of the player so that the sound reproduction is not good enough for your machine. You might not notice the difference when playing music, but your Oric certainly will. On the other hand, don't be taken in by advertising and buy 'computer quality' tapes. These are overpriced and a standard C60 will serve you far better.

You may find that you should keep your tape deck as far away as possible from both the TV and the Oric in order to avoid electrical interference when CSAVEing or CLOADing programs. Similarly, depending on your tape deck, you may find that it's better to run the deck off batteries, as it may pick up some mains hum if run from an internal transformer via a mains supply.

Sometimes you may find that a program won't RUN after it seems to have CLOADed correctly. If you LIST the program and screenfuls of the letter U appear, all you have to do is to enter a dummy line like 1 PRINT, then delete the line. This should sort out the problem, which is caused by corrupted pointers. Adding a new line forces the Oric to sort these out. If it doesn't, you'll just have to re-CLOAD the program at a lower volume. You will probably find that it takes a bit of trial and error to establish the correct tone and volume settings of your cassette to get reliable loads. When you've found these, mark the controls with Tippex or something, for reference. If programs then suddenly won't CLOAD, you'll probably need new batteries.

As a final point, keep a careful index on each cassette of every program you CSAVE, together with such information as which speed was used to CSAVE it and a counter number as a rough guide to its location on the tape if your deck has that facility. This will save you hours.

The Oric's (version 1.0) file-handling system is very poor compared to just about any other machine available on the market. You can only save programs or blocks of memory to tape. Other machines allow you to save the data in arrays, open sequential text files for input or output and so on. The Oric (version 1.0) offers no facility for saving data to tape.

The routines for saving information to tape must exist in the ROM as they

are used to save programs. *Oric Owner*, a magazine produced by Tansoft, published a machine-code routine by Paul Kaufman (Issue 2, June–July 1983) which will allow you to save data to cassette, but it seems incredible that you should have to enter such a routine as a piece of external code when it should have been included in the ROM (version 1.0).

More programs for you to write

To give you a chance to tackle some problems in programming that are rather more involved, here are two suggestions. First, Compound Interest. Write a program that will work out how much money you would have in a savings account after a specified number of years, at a specified rate of interest, leaving the interest in all the time to accumulate. Second, the Birthday Problem. What is the probability that in any given room full of people, two of them will have the same birthday? You will find it is surprisingly high, and that with as few as twenty-three people present, the probability is over 0.5 (even odds). Both of these problems are extremely tedious to work out in a conventional way, even with a pocket calculator.

Suppose for your savings account you are receiving 12 per cent interest. At the end of the first year, your money is multiplied by 1.12, making it slightly larger. Then at the end of the second year that increased money is multiplied by 1.12, making it larger still, and so on. The formula for compound interest is:

$$A = P \times \left(1 + \frac{I}{100}\right)^N$$

where P is the principal (the money you start with), I is the percentage rate of interest, N is the number of years the money is left in the account, and A is the accumulated total. For I equals 12 per cent the formula looks like:

$$A = P \times (1.12)^N$$

in which 1.12 is raised to the power of N. In other words, if N is 3, we have $P \times 1.12 \times 1.12 \times 1.12$. In Basic, powers are indicated by an up arrow ∧ (SHIFTed 6) so if you give the command, say,

```
PRINT 5∧2
```

you get

```
25
```

which is 5 × 5. Try giving the command 'PRINT 4∧3'. You should get 4 × 4 × 4 which is 64.

Write a Basic program that asks for P and N and then works out the accumulated total at 12 per cent interest. When that is working, you could elaborate it to ask for I as well, so that it can be used for any rate of interest, and you could also allow for interest being added every six months rather than just once a year.

When it is working, test your program with plenty of made-up examples. In 1626 the Indians sold Manhattan Island to the Dutch for $24-worth of beads. It is sometimes said that if they had invested the $24 then at compound

interest, they would have as much money now as the island's present value. Is this true?

To solve the Birthday Problem, you call the number of people in the room N and work out this formula:

$$P = 1 - \Bigg(\underbrace{\frac{365}{365} \times \frac{364}{365} \times \frac{363}{365} \times \frac{362}{365} \times \frac{361}{365} \cdots}_{\text{N terms}}\Bigg)$$

You can see how tiresome this would be to work out with a hand calculator for, say, twenty people. The formula is derived from the idea of working out the probability that no two people have the same birthday, and then subtracting from 1. In your Basic program, it is best to work out all the multiplications first, and then subtract from 1. You will need to set up a loop which counts terms as it goes around, and subtracts 1 from the numerator each time as well.

The new ROM

At the time of writing, Oric International Ltd are in the stages of producing a new ROM chip for the 'new' Oric computer, the Atmos. This will contain version 1.1 of the Basic interpreter, with improved file-handling and fewer bugs. This section details the most important changes made.

Verifying programs

With the new ROM, the computer will be able to tell you whether a program you have just saved on to tape has been recorded properly. This process is known as verification. To get it to do this, you would first CSAVE the program as usual, then rewind the tape to the beginning of the program. Now enter 'CLOAD "*program name*",V' (you would also have to add ,S if the program was saved at slow speed). The micro will now look for the program on the tape. If it finds any others it will display the message 'FOUND . . . *program name* B' on the status line. When it finds your program, the message 'VERIFYING . . . *program name*' will appear. It will now compare what's on tape with the program in memory. If the program has been recorded properly, you should get the message '0 VERIFY ERRORS DETECTED'. If this figure is not zero, then you'll have to try saving the program again.

Merging programs

A nice feature of the new ROM is the ability to join or merge a program on tape with one in memory. This means that you could write a set of subroutines and save them to tape, then add them to every program which needed them, without having to type them in all over again.

To use the merge facility, enter 'CLOAD "*program name*" ,J' (J for join). The Oric will display the usual file-handling messages. The main thing to be careful with here is to make quite sure that the program you're adding has line numbers that are *totally* different from the program in memory. This means that

you might have a program on tape whose first line number was 20000, and merge it to one in memory whose highest line number was well below this, or vice versa. If you do not do this, the two programs may well become very confused in memory and the resultant program may not work properly.

Data

With version 1.1 you can CSAVE the contents of string, integer or numeric arrays, then reload them in different parts of the same program that saved them, or in a different program. In the latter case, the array must have been DIMensioned to the same size as the original.

To save an array, simply use the command 'STORE A, "*array name*" ' (together with the ,S if you wish). This will give one of three very similar messages at the top of the screen: for example, 'SAVING . . . *array name* I'. The last letter may be I (integer array), S (string array) or R (real, i.e. numeric).

To load the array, you use the new word RECALL, as in 'RECALL A$,S', which would look for a string array called A$, saved at slow speed. As usual, the messages 'FOUND . . . *file name*' or 'LOADING . . . *file name*' will appear at the top of the screen.

If the message 'ERRORS FOUND' appears, the array has not been loaded properly and while your program will still work, the information in the array will not necessarily be the same as what was saved in the first place. You would be well advised to offer a user the chance to try reloading the array if this occurs within a program, by directing the program back through the same loading routine until the user was happy with the load.

Print@

The new ROM has a version of PRINT@ found on most other micros. This command has to be followed by two numbers separated by a comma. These are the column and row numbers where you want a message to begin. The command finishes with a semicolon just before the item to be PRINTed. For example:

```
100 PRINT@ 10,12;"MESSAGE":REM display
    "Message" starting at column 10 of row 12
```

Note that PRINT@ allows you to PRINT in column 0, the normally protected column used for the PAPER colour of the screen.

Suggested answers for 'instructions' exercises

You may well disagree with these!

Filling a fountain pen
Get pen
Get ink bottle
Remove pen cap
Remove bottle cap
Dip nib in ink
Pull down lever
Raise lever
Pull down lever
Raise lever
Remove nib from ink
Wipe nib
Replace pen cap
Replace bottle cap

Making a pot of tea
Fill kettle
Light gas
Put kettle on gas
Wait until boiling
Turn off gas
Get teapot
Pour some hot water into pot
Swill around
Pour away
Put one spoonful of tea for each person into pot
Put one spoonful of tea in for pot
Pour in water
Wait until brewed

3. How the Oric Works Inside

So far you have read a great deal about what your computer does, but nothing about how it works. As it happens, it is possible to understand computers on a number of different levels; so, for instance, one can use them very effectively while having virtually no idea how they work. At the same time, using to the full some of the advanced facilities of your Oric requires some understanding of what is going on inside. This does not, however, involve knowing about the lowest level of detail – that is very complicated and nearly everybody leaves that to specialist engineers. This chapter explains the principles of how the Oric works inside, to strip away some of the mystery of the technology and to enable you to use many of the extra facilities and power of the machine.

Any computer in operation has two distinct parts: the physical computer itself – electronics, keys, screen, etc. – and the program which you or someone else wrote. Both of these are essential for the computer to work. The physical parts are often known collectively as 'hardware', while programs, parts of programs and collections of programs are called 'software', because of the ease with which they can be changed. The logical structure of the hardware of your Oric is shown in Fig. 3.1.

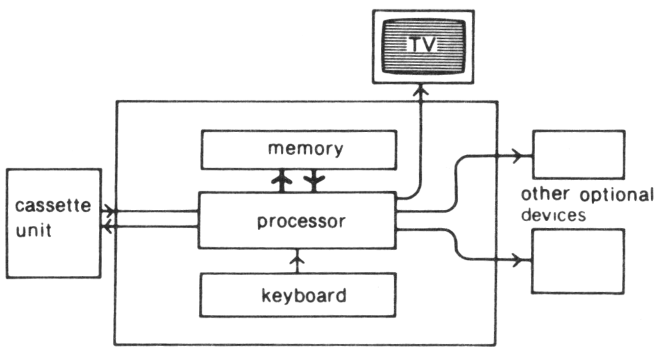


Fig. 3.1 Schematic layout of the Oric

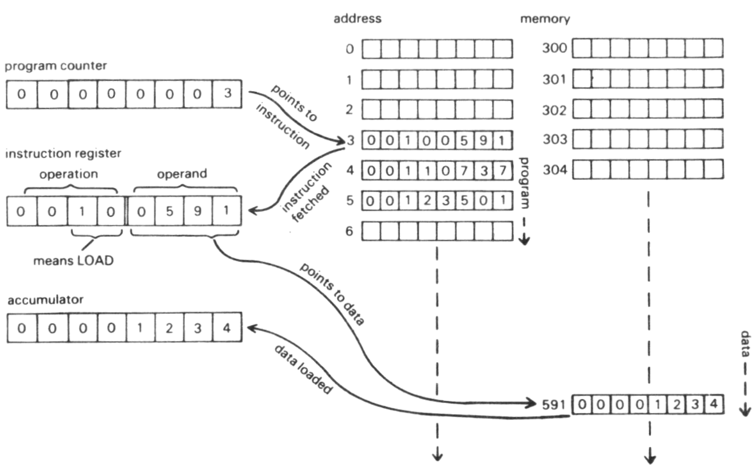
Inside, the machine works entirely with *numbers*. The instructions are numbers, the data are numbers, and places in the memory are located by numbers. The Basic words you type in are translated into these numbers, and then executed. This process is very complex, so the best way to get an idea of what goes on is to examine the workings not of an Oric but of an imaginary computer, much simpler than the Oric but operating on exactly the same principles.

Within any computer, numbers are held in electronic circuits called *registers*. The physical size of a register determines the largest number it can hold: most

registers in the Oric can hold up to 255, while larger computers' registers go up to 65,535 and beyond. (The reason for these curious numbers will become apparent later.) Within the processor of our imaginary computer there are three special-purpose registers: the Program Counter, the Instruction Register and the Accumulator. The Program Counter controls the flow of the program, the Instruction Register deciphers the instructions, and the Accumulator is where the data are actually processed.

The computer's *memory* consists of several thousand identical registers or 'cells'. These are numbered sequentially so that the processor can refer to them – each cell's number is called its 'address', as a natural word for specifying location. All this is shown in Fig. 3.2. The memory holds both the program and the data on which it is to work. These are indistinguishable, so the computer has to keep track of where the program is and where the data are. Each instruction in a program, like everything else, is a number, and the program occupies a series of consecutive cells in memory.

The Program Counter contains the address of the instruction to be executed next. The processor goes to that cell in the memory, reads the number contained there and loads it into the Instruction Register. The instruction is carried out, the address in the Program Counter is increased by one, and the process is repeated with the next instruction of the program. The machine thus works in *cycles* of two parts: the first half called Fetch in which the instruction is read out of memory, and the second half called Execute in which the instruction is carried out. One cycle typically takes about a millionth of a second.

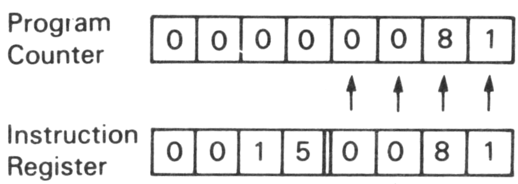


Instructions:
10 Load
11 Add
12 Store
13 Subtract
15 Jump
16 Jump if Accumulator is zero
etc.

Fig. 3.2 How a computer works internally

The instruction as it sits in the Instruction Register has two parts: an *operation* and an *operand*. We can see these as corresponding to the verb and the object in an English sentence (all instructions are imperatives so there is no subject stated). Each instruction says, 'Do *this* operation to *this* piece of data.' Suppose the operation is the number 10 and the operand the number 0591. Operation 10 could mean 'Load'. Thus the whole instruction means: 'Read the number stored in memory cell 0591 and put it in the Accumulator.' Operation 11 could mean 'Add'. Then instruction 11-0737 would mean: 'Add the number that is in memory cell 0737 to the number that is currently in the Accumulator and put the result in the Accumulator.' Instruction 12 could signify 'Store' and 12-3501 would say: 'Take the number that is in the Accumulator and store it in memory location 3501.' (Whatever was previously in that location gets thrown away.) The processor would have a dozen or so major instructions like these, including 'Subtract' and 'Clear the Accumulator to zero'. Some simple computers, including the Oric, do not even have a 'Multiply' instruction – they require the programmer to write a short routine to multiply by repeated addition.

It is important to see that these instructions are not the same as Basic commands, nor are the memory addresses the same as Basic line numbers. All your Basic commands get translated by the computer into these numbers. To move about in the program in the manner of a Basic GOTO, the computer has a 'Jump' instruction. If the 'Jump' is instruction code 15, a whole instruction could look like: 15-0081, meaning 'Carry on executing the program from memory cell 0081.' The mechanism for this is very simple: the processor just takes the operand part of the instruction (which is the address to be jumped to) and forces it into the Program Counter, like this:



This causes the program to carry on from that point.

For making decisions, there is a *conditional* Jump instruction, saying, 'Jump to the address indicated *if the Accumulator is zero*.' This way a question box on a flowchart can be implemented by having a test on a particular piece of data: if it is zero, the program jumps; if it is not, the program carries on in the normal sequence of steps. All questions have to be reduced to the form: 'Is something or other equal to zero or is it not?' Thus to ask 'Is number A equal to number B?' the program has to subtract one number from the other and test to see whether the result is zero.

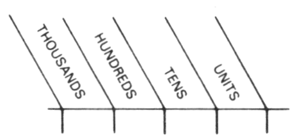
There has to be one other set of instructions and that is for input and output. One instruction would mean: 'Accept a character that has been typed on the keyboard and load it (yet again encoded as a number) into the Accumulator.' Another command could be: 'Take the character held in the Accumulator and send it to the screen.' All other input and output devices such as the cassette unit have their own Input/Output commands.

The instructions that have been described here are really very crude – each one achieves very little. However, by putting together long sequences of these crude instructions it is possible to make the computer perform very complex tasks. A great many of these very small steps are involved in any useful operation. As a result, even though one millionth of a second may seem a miraculously short time for one processor cycle, it is none too fast when it comes to tackling real jobs. It probably will not be long before you give your Oric a program that is complicated enough for there to be a noticeable delay before you get the answer.

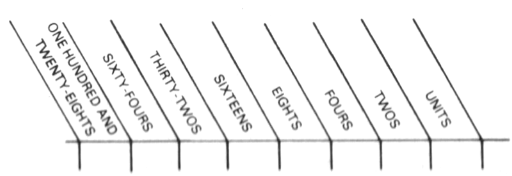
The numerical instructions make up what is known as the computer's 'machine code'. It is possible to program the machine entirely in these numbers, but it is extremely tedious, as numbers are hard to remember. This is why computer languages such as Basic have been devised. The English words of the language are translated into the machine code by a special program called an *interpreter*. This is held in the Oric in a special section of memory that cannot be erased, called ROM for 'read-only memory'. (You can only 'read' data from it – you cannot 'write' data into it.) Each Basic command is translated typically into several dozen machine-code instructions. For example, one PRINT command will carry out a calculation and then cause a whole series of characters to appear on the screen, while one machine-code output command will transfer only one character.

Despite the difficulty of programming in machine code, there are purposes for which it is useful to be able to do so, rather than using Basic. Machine code is much more flexible than Basic, and programs in it run faster. If you are interested in doing this, you will need to consult a 6502 programmer's reference guide, as it is outside the scope either of this book or the standard Oric manual.

Having seen how the computer functions in terms of numbers, we shall next look at how the machine actually works physically. Since everything inside the machine is a number, the first question is how numbers can be represented in some physical form. The earliest electronic computers used varying voltages of electricity to do this: the stronger the voltage, the larger the number. This method of working, called 'analogue', was awkward and unreliable and was soon superseded by 'digital' computers which use circuits that are either on or off – there is nothing in between. To use these on–off circuits to hold numbers, we need to count in twos, rather than in tens as we do normally. This is not as odd as it may seem. Instead of the digits being labelled:



they are:



This is called the 'binary' system, as opposed to the usual 'decimal' system. This is what the decimal numbers from one to ten look like in binary:

ONE HUNDRED AND TWENTY-EIGHTS	SIXTY-FOURS	THIRTY-TWOS	SIXTEENS	EIGHTS	FOURS	TWOS	UNITS	
							1	= 1 decimal
						1	0	= 2
						1	1	= 3
					1	0	0	= 4
					1	0	1	= 5
					1	1	0	= 6
					1	1	1	= 7
				1	0	0	0	= 8
				1	0	0	1	= 9
				1	0	1	0	= 10
7	6	5	4	3	2	1	0	← 'BIT' Numbers

As one example, 101 in binary is

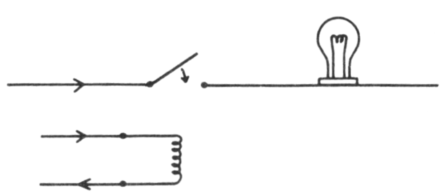
1 four
0 twos
and 1 unit

---

making 5 in decimal

One binary digit (0 or 1), also called a 'bit', is the smallest possible unit of information. If a register is eight bits long, the largest number it can hold is 11111111 or 255 in decimal. A sixteen-bit register can hold 1111111111111111 or 65,535.

The effect of counting in binary is that numbers are much longer than in decimal, but only two symbols are needed (0,1) instead of ten (0,1,2,3,4,5,6,7,8,9). 0 can easily be represented by 'Off' and '1' by 'On'. A digital computer therefore is nothing but an assemblage of on–off switches. The switches, though, cannot be activated by a human being's finger like household switches – the machine has to be automatic, so it must be possible for the switches to be turned on and off *by other electric currents.* Such devices have existed for many years and they are known as 'relays'. These consist of a switch with the coil of an electromagnet alongside:



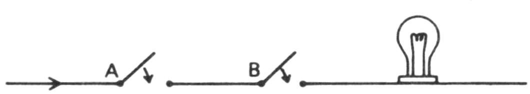
When current flows through the coil it becomes magnetized, the arm is pulled down and the switch goes on. At first, digital computers were built with relays, but these are slow and prone to mechanical failures, so they were soon replaced by electronic valves (vacuum tubes). These have no moving parts but they have the drawbacks that they are expensive, take up a lot of room and generate a great deal of heat. The computer did not really become a practical proposition

until the invention in the late 1940s of the transistor, in which the electric charges themselves act as a switch, using a microscopically small junction in pieces of exotic materials called semiconductors. From being built out of separate transistors soldered together, computers have progressed to the stage where entire processors consisting of thousands of transistors are made all in one piece on a single sliver or 'chip' of the element silicon. These 'integrated circuits' as they are called can be mass-produced very cheaply and it is this that has brought about the microelectronic revolution of recent years. The central processor of your Oric is an example of an integrated circuit, specifically an LSI chip, for 'Large-Scale Integration' and is called a 6502.

Until fairly recently computers had memories made out of thousands of tiny magnets – magnetization in one direction meant 0 and in the other direction meant 1. Nowadays it is cheaper to use transistor circuits which are off for 0 and on for 1; a separate circuit is required for each bit stored. These have the disadvantage, as you will already be aware with your Oric, that information in a memory is lost when the power is shut off; hence the need for cassettes and other storage devices.

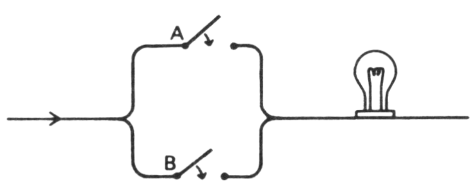
Central to the workings of the switches in a computer is a system of logic invented in the 1850s by George Boole. In this, all statements have one of two values: True or False, and there are three connectors: AND, OR and NOT. These enable one to construct descriptions of the world such as: 'We will play football tonight IF there is nothing good on television AND the weather is fine.' The truth of the statement 'We will play football tonight' can be established by finding out the truth of the two following clauses and linking them with the logical AND operation. OR works like this: 'We will have a goalkeeper IF Bill turns up OR Bob turns up.' It is of course possible for both of them to turn up, and OR is taken to include that possibility. Finally, NOT: 'We will play IF the pitch is NOT taken by somebody else.'

These can easily be implemented in electronic circuitry. True corresponds to On or 1 and False to Off or 0. AND is two switches in series:



The light will go on only when *both* switch A *and* switch B are on. An arrangement of switches like this is called an 'AND gate'.

OR is two switches in parallel:

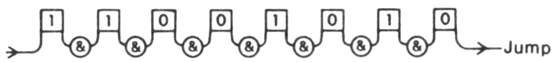


The light will go on if either switch A *or* switch B is on, or both. This is an 'OR gate'.

NOT is a switch with a contact on the 'released' side:

When the lever is pushed down, the light goes *off*.

Interestingly enough, virtually all the functions required in a digital computer can be implemented using combinations of these three types of gate. As a single example consider the instruction 'Jump if the Accumulator is zero.' The Accumulator consists of one circuit for each binary digit (eight in the Oric). For the condition for a jump to be fulfilled *all* the digits have to be zero, so for this instruction all the circuits are connected together in a series of NOTs and AND gates. Only if the digits are all zero will current pass through the AND gates and activate the 'Jump' circuitry.



Each of the other functions of the processor has its own set of circuits that are brought into play when the appropriate instruction is in the Instruction Register: one set for Loading, another set for Adding, and so on. It is easy to see from this why a practical computer needs many thousands of circuits.

To give you a taste of machine code in action, here's a routine which scrolls the entire TEXT screen to the left by one cell each time it's called. The routine can only be used within a program and you have to 'POKE 49120,32' before issuing a 'CALL 1024' to get it going.

```
 1Ø DATA 48,8A,48,98,48,AØ,1B,A9,AA,85,ØØ,A9,
    BB,85,Ø1,A2,26
 2Ø DATA 84,Ø2,AØ,Ø1,B1,ØØ,AØ,ØØ,91,ØØ,2Ø,32,
    Ø4,CA,DØ,F2
 3Ø DATA 2Ø,32,Ø4,2Ø,32,Ø4,A4,Ø2,88,DØ,E3,68,
    A8,68,AA,68,6Ø
 4Ø DATA 18,A5,ØØ,69,Ø1,85,ØØ,A5,Ø1,69,ØØ,85,
    Ø1,6Ø
 5Ø DATA Z
 6Ø CS=Ø:AD=1Ø24
 7Ø REPEAT
 8Ø READ A$
 9Ø V=VAL("#"+A$): CS=CS+V: REM CS is check sum
1ØØ POKE AD,V
11Ø AD=AD+1
12Ø UNTIL A$="Z"
13Ø IF CS<> 5954 THEN CLS: PRINT "DATA ERROR":
    LIST-4Ø
14Ø CLS:DEF FNR(N)=INT(RND(1)*N)+1: PAPER 2
15Ø FOR I=1 TO 5Ø
16Ø PLOT FNR(38),FNR(26),FNR(26)+64
```

```
170 NEXT
180 FOR I=1 TO 38
190 POKE 49120,32
200 CALL 1024
210 NEXT
```

The program is in three parts: the first consists of the machine-code program in hexadecimal notation. The numbers in the DATA statements consist of instructions and numbers to be used in the instructions. For example, the sixth datum is an instruction to load the Y register with the next number (1B: 27 in decimal) which is the number of lines to be scrolled.

'Hexadecimal' is a way of counting in base 16. It is used to make programming in assembly language easier. Numbers run as in base ten from 0 to 9, but ten is A, fifteen is F, 255 is FF and so on. You can find out the 'hex' for a decimal number using HEX$: for example, 'PRINT HEX$(1024)'. If you precede a number with a hash sign (#), the Oric will assume it to be a hex number: for example 'PRINT #400'.

The following shows how the machine-code mnemonics 'translate' into Basic:

Basic	Machine-code Mnemonics	Comments
10 Y=27	LDY,1B	load Y with 27
20 REM scroll row routine	SCR;	scroll row routine
...	...	...
80 Y=Y-1	DEY	decrement Y
90 IF Y>0 THEN GOTO 20	BNE SCR	branch back to scroll

The second part is a simple 'hex loader', which READS the hex numbers as strings, converts them into decimal (by using VAL and the hex symbol #), then POKEs the numbers into page 4 of RAM, which is reserved for users' machine-code routines. Page 4 starts at address 1024. There is a 'check sum' included which adds up the values. If you've made a mistake in the DATA statements, the program will stop and LIST the DATA statements so that you can edit them. This is necessary because a mistake in a single character could cause Oric to 'crash' or 'hang up', i.e. refuse to respond to the keyboard, the only cure being to unplug and restart the machine. The third part of the program puts some random characters on the screen, then demonstrates the machine-code routine by CALLing it.

You can alter the amount of screen which scrolls in two ways. Firstly, you can change the number of lines which are scrolled. The number to change is held in the seventh DATA item. In the program it is set to 1B (27 in decimal). This is because there are twenty-seven screen rows (see Chapter 6). If you want to have only the first ten rows scrolled, you could alter the seventh DATA item to A (10 in decimal). This will, however, mean that you will have to alter the check-sum routine in line 130. The best way to do this is simply to take it out, since by now you should have a working program anyway. Another

way of doing this is to 'POKE 1030,10'. This method is preferable because it means that you don't have to rerun the loading program.

Another change you can make is to alter the values held in the ninth and thirteenth DATA items. These control where the routine starts scrolling from. You can set these values so that only the lower part of the screen scrolls. For example, to have only the last 10 rows scroll you would have to start the scrolling from the sixteenth row. This starts in memory at address 48040 + (40 × 16) + 2; Chapter 6 gives more details. To get this number into the routine you first have to convert 48682 to hex, which is BE2A. Then you take the first two letters or numbers (BE) and put them at the thirteenth DATA position. The remaining hex number (2A) goes at the ninth position. Alternatively, you can 'POKE 1032, 42: POKE 1036, 190'.

Easier still, set the variable RO to a value between 1 and 26, then call the following subroutine with a GOSUB 9000.

```
9000 AB=48042+(40*RO)
9010 A$=HEX$(AB)
9020 L$=MID$(A$,2,2): R$=RIGHT$(A$,2)
9030 LV=VAL("#"+L$): RV=VAL("#"+R$)
9040 POKE 1032,RV:POKE 1035,LV
9050 RETURN
```

4. The Basic Language in More Detail

Up to now, all the processing your computer has done for you has been of *numbers.* Doing mathematics is what computers are best known for – indeed, that is where they get their name – but they are equally important for processing *words.* You have, of course, already made your Oric display words, as in the command 'PRINT "HOW DO YOU DO?"' but there has been no actual processing involved. To carry out operations on words and groups of words, you need to be able to refer to them in the same way as you do numbers, that is, with labels such as A, B, C.

The numbers that you have been storing in memory with labels are known formally as 'variables'. To handle words, Basic allows you to create what are called 'string variables'. A string is simply any series of characters: letter, figures, punctuation marks or spaces.

```
GRQ ##3£ ,SY9 ?Z
```

is a string, as is

```
TO BE OR NOT TO BE
```

Both are equally meaningless to the computer; to it they are just strings of characters. To store some words in memory you use the LET command as with numbers, *except that the label you give must have a dollar sign on the end.* The words must be enclosed in quotes as in a PRINT statement:

```
LET A$="GOOD MORNING"
```

Then if you say

```
PRINT A$
```

the machine should reply

```
GOOD MORNING
```

The string can be of any length up to 255 characters. The dollar sign as part of the variable name is essential because strings and numbers are held in memory in quite different forms. The machine will not allow you to mix numerical variables and string variables. If for instance you say

```
LET X=A$
```

you will get an error message. Similarly

```
LET N$=X
```

will not work, while

```
LET N$="X"
```

will work. Do you see the importance of the quote marks? In the last case N$ becomes a string containing just the one character X.

The first way strings can actually be processed is in IF statements. Look at this simple program:

```
10 PRINT "GIVE THE PASSWORD."
20 INPUT B$
30 IF B$="OPEN SESAME" THEN GOTO 60
40 PRINT "WRONG. YOU SHALL NOT PASS."
50 END
60 PRINT "PASS, FRIEND."
```

When it encounters the INPUT command the computer prints a ? as usual, and the user types in whatever he or she thinks the password is, finishing it with RETURN (no quotes are necessary here). At line 30 the crucial comparison is made; only those exact words will be accepted by the computer. Even an extra space will cause the comparison to fail. An extra refinement to this program could be that instead of just saying PASS, the Oric could actually operate the lock on a door, using a special attachment.

Here is another example of a program using string variables:

```
10 PRINT "ANSWER YES OR NO."
20 INPUT A$
30 IF A$="YES" THEN GOTO 70
40 IF A$="NO" THEN GOTO 70
50 PRINT "STOP MESSING ABOUT."
60 GOTO 10
70 PRINT "O.K."
80 GOTO 10
```

This is a striking example of how it is possible to make a computer appear to converse in an almost human way. If in reply to its 'ANSWER YES OR NO', you reply 'DROP DEAD', its response 'STOP MESSING ABOUT' seems very natural. But of course it does not actually *understand* what you say; all it knows is that it is not either YES or NO. AFFIRMATIVE will be rejected just as sharply.

Names of variables

Up to now we have used only single letters (with or without $) to label variables. In fact, Basic allows you to use whole words, and this can be useful in making it easier to remember what each variable is and generally making your programs more comprehensible to humans. Thus instead of saying, in the program for calculating averages (p.25):

```
20 LET S=0
```

you can say

```
20 LET SUM=0
```

Throughout the program you refer to that variable as SUM rather than S. Likewise C could be called COUNT. String variables can also have longer names:

```
LET NAME$="ANGELA"
```

One thing you need to remember about this is that Basic only pays attention to the first *two* letters. Thus, as far as Basic is concerned, JUDY and JULIA are exactly the same, and it will treat them as the same variable (JU) in a program. Figures can also be used in variable names, except that the first character must always be a letter. This way you can have variables N1 and N2 for example, but not 1F, 9B, etc.

Another point is that Basic command words cannot be embedded anywhere in a variable name. Basic generally ignores spaces (at least those not inside quote marks) so it sees no difference between GRIFFIN and GR IF FIN. It will assume that IF here is a command word and will try unsuccessfully to make sense of the line on that basis. This means that GRIFFIN cannot be used as a variable name. Nor, similarly, can PALLET or FENDER.

In other words, although you have a wide variety of possible variable names to choose from, such as AZ, N1, PQ, etc., the Oric will not let you use some combinations. Try the following:

```
10 FOR GO=1 TO 10
20 NEXT
```

You will get a '? SYNTAX ERROR' for line 10. This is because the two letters G and O together form part of one of the Oric's 'reserved words'. These are Basic words which the Oric recognizes as having special meanings.

The Oric will not tolerate any variable names which bear a close resemblance to any of its reserved words. GO is the first part of both GOTO and GOSUB, so cannot be used. Other illegal combinations include: GO, TO, FN, ON, IF, LN, OR and PI.

FOR . . . NEXT

Before you have written many programs you will find yourself frequently having to set up loops with counters in them. Each time round the loop your program adds one to the counter, tests to see whether it has reached a required total or not, and loops back, or not, as the case may be. This is such a common requirement in Basic that a special command is provided to reduce the number of instructions you have to write. This is the word FOR. With the command you name a variable which you want to take on not just one but a series of different values. The instruction looks like this:

```
FOR N=1 TO 10
```

This causes the computer to set N equal to 1 and then carry on with the following instructions in the program, *until it encounters the command* NEXT. This makes it go back to the FOR instruction, add 1 to N, and go through the same instructions as before. This carries on until N equals 10. As an example,

suppose you want to print the word HELLO ten times. The old way of doing it would be like this:

```
1Ø LET COUNT=Ø
2Ø PRINT "HELLO"
3Ø LET COUNT=COUNT+1
4Ø IF COUNT<1Ø THEN GOTO 2Ø
```

With FOR . . . NEXT it can be done like this:

```
1Ø FOR COUNT=1 TO 1Ø
2Ø PRINT "HELLO"
3Ø NEXT COUNT
```

Try this for yourself. Suppose you accidentally stand on someone's foot and you say, 'Ooh. A thousand pardons', and they reply, 'OK. Let's have them. The thousand pardons.' Instead of saying 'Pardon me pardon me pardon me . . .', get your Oric to do it!

As an example of how the variable in the FOR statement can actually be involved in a computation, here is a way of printing out a multiplication table. In this case it is the seven times table:

```
1Ø FOR N=1 TO 12
2Ø PRINT 7*N
3Ø NEXT N
```

On RUN this gives:

```
7
14
21
28
35
42
49
56
63
70
77
84
```

To make the output more meaningful, line 20 could be:

```
2Ø PRINT N " TIMES 7 MAKES" 7*N
```

Try it. You could also try other tables than seven.

You can get even more clever and make the computer display all the times tables from two to twelve with one program. This involves putting one FOR loop inside another – this is called 'nesting' loops. The easiest way of seeing how this works is by looking at the program:

```
10 FOR M=2 TO 12
20 FOR N=1 TO 12
30 PRINT N " TIMES" M " MAKES" N*M
40 NEXT N
50 PRINT
60 NEXT M
```

The arrows show you how the inner N loop goes round on itself, and the outer M loop does likewise. Each pass round the M loop involves twelve passes round the N loop.

The N and M at the end of the NEXT commands are not strictly necessary for the computer, but they help you keep track of what is going on. The PRINT command with nothing after it at line 50 inserts a blank line in between each table, to make them easier to read. Of course, the tables run off the top of the Oric screen faster than you can read them properly, but you get the idea! You can stop the program by pressing CTRL C.

Sometimes you have a loop in which the variable needs to be increased by something other than 1, each time round. In this case you use the word STEP at the end of the FOR instruction, followed by a number giving the required increment. This can be a whole number or a fraction, or it can even be negative. Suppose you want a table showing British money converted to American, running from 1p to £5.00, in steps of 1p (£0.01). Assuming the exchange rate is $1.45 per £1, the program looks like:

```
10 FOR L=0.01 TO 5 STEP 0.01
20 PRINT L;" POUNDS=$";L*1.45
30 NEXT
```

FOR loops are sometimes used to do nothing at all – just to waste time. In doing animation on the screen for instance, it is often necessary to draw something, wait a fraction of a second, then draw it again in a slightly different position, and so on. The instructions

```
FOR N=1 TO 500
NEXT
```

will put a short but noticeable delay into a program while the computer goes round the FOR loop five hundred times. Changing the number obviously changes the length of the delay.

You could use the WAIT instruction to do this as well. WAIT 10 causes a delay of 100 milliseconds. WAIT 100 gives a pause of 1000 milliseconds (about one second).

Subroutines

Another important idea in programming is that of a *subroutine*. It often happens that there is a particular set of instructions, usually short, that has to be carried out in exactly the same way at several different points in a program. Take, for instance, a game program in which a little figure of a man has to be drawn, but in different places on the screen. It might take two or three lines to draw the figure. These same instructions could be written into the program everywhere

they occur, but this would be tedious and wasteful of memory space. A better technique is for those instructions to be put aside in a special place in the program, constituting a *subroutine*. Then, wherever in the main body of the program those instructions need to be, a single GOSUB (Go to Subroutine) command is used. This acts like a GOTO when it is encountered in the running of the program, but at the end of the subroutine is the single word RETURN. This makes the program jump back to the instruction following that GOSUB. This can best be seen from the example:

```
  1Ø HIRES
  2Ø LET C=9Ø: LET R=5Ø
  3Ø CURSET C,R,Ø
  4Ø GOSUB 1ØØØ
  5Ø CURMOV 6Ø,Ø,Ø
  6Ø GOSUB 1ØØØ
  7Ø CURMOV Ø,6Ø,Ø
  8Ø GOSUB 1ØØØ
  9Ø CURMOV-6Ø,Ø,Ø
 1ØØ GOSUB 1ØØØ
 11Ø END
1ØØØ CIRCLE 5Ø,1
1Ø1Ø RETURN
```

Each time a GOSUB is encountered, the program jumps to 1000 *remembering where the GOSUB was.* Then when RETURN is reached, the program jumps back not to one particular point in the program but to the command after the command which 'called' the subroutine. This is a very useful facility for breaking down a complex operation into a series of smaller tasks, each of which can be made into a subroutine, entirely self-contained. Then the overall job of writing the program can become one of assembling the appropriate subroutines, each of which acts like a single Basic command, since it can be called by one GOSUB instruction.

You can make this program more interesting if you change line 1000 to '1000 CIRCLE 50, FB,' change line 110 to '110 IF FB = 1 THEN FB = 0 ELSE FB = 1', and add '120 GOTO 30' and 15 'FB = 1'.

These alterations will draw the circles, then erase them, until you break out of the program using CTRL C.

POP

POP is a command that drops a GOSUB label off the stack of RETURN addresses. That is, it can be used to leave a subroutine and RETURN, not to the routine which called it, but to the routine which called the latter as a subroutine. POP is useful when you have a number of levels of subroutines in a program – where one subroutine calls another with a GOSUB, and this subroutine may call yet another subroutine and so on.

For example, you might have a main menu which can GOSUB to one of three subroutines. Each of these may GOSUB to other subroutines. To get

back to the main routine from any of these latter subroutines, you would normally have to RETURN to the second-level subroutines, and from there RETURN to the main routine. POP allows you to do away with such round-about techniques. At the lower levels mentioned above, should you want the program to RETURN to the main routine, all you have to do is issue a POP: RETURN command, which will return control to the level above the level that called the subroutine.

A tree diagram of this might look something like Fig. 4.1.

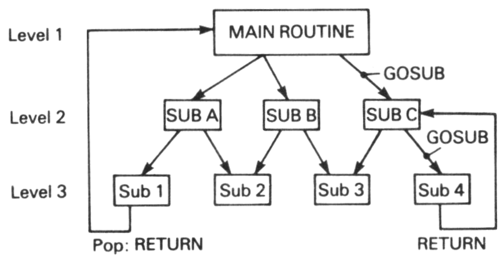


Fig. 4.1 Using POP

As an example of POP, try the following:

```
 1Ø CLS
 2Ø PRINT "PRESS KEY WHEN READY"
 3Ø GET A$
 4Ø GOSUB 1ØØ
 5Ø GOTO 1Ø
1ØØ PRINT "SUBROUTINE AT 1ØØ"
11Ø PRINT "PRESS 1 TO RETURN TO START, 2 TO
    GOSUB 2ØØ"
115 GET A$:A=VAL(A$):IF A<1 OR A>2 THEN GOTO 115
12Ø IF A=1 THEN RETURN
13Ø GOSUB 2ØØ
14Ø GOTO 1ØØ
2ØØ PRINT "SUBROUTINE AT 2ØØ"
21Ø PRINT "PRESS 1 TO RETURN TO START, 2 TO
    RETURN TO 1ØØ"
22Ø GET A$:A=VAL(A$):IF A>2 OR A<1 THEN GOTO 22Ø
23Ø IF A=1 THEN POP:RETURN
24Ø RETURN
```

You can see from using this trivial example how you might use POP to get back to higher levels of a program with many GOSUBs without having to RETURN through each subroutine. POP allows you to jump out of nested subroutines without needing to pass extra variables about to handle all the RETURNs. It means that you don't have to match every GOSUB with a RETURN.

REPEAT

As well as FOR ... NEXT loops, the Oric uses a REPEAT ... UNTIL construct. In some applications, where you don't necessarily want some action performed a set number of times and where you may wish to have the loop terminated by a potentially unpredictable event like a key-press, REPEAT ... UNTIL is ideal.

The general form of this loop is: 'REPEAT *actions* UNTIL *condition*(*s*)'. For example, you could use this construct to make a subroutine which would wait until the space bar was pressed like this:

```
1000 PRINT "PRESS THE SPACE BAR TO CONTINUE"
1010 REPEAT
1020 LET A$=KEY$
1030 UNTIL A$=CHR$(32)
1040 RETURN
```

This gives exactly the same result as:

```
1000 PRINT "PRESS THE SPACE BAR TO CONTINUE"
1010 LET A$=KEY$:IF A$<>CHR$(32) THEN GOTO
     1010 ELSE RETURN
```

However, the first is easier to read and if the loop structure is at all complex, is preferable to the latter.

PULL

A nice feature of REPEAT ... UNTIL loops is that you can terminate a loop using the command PULL. This acts rather like setting the index in a FOR ... NEXT loop to, or above, the upper limit (a method for jumping out of such loops which avoids many problems). In effect PULL does the same thing by ending a REPEAT ... UNTIL as if the condition in the UNTIL statement had been fulfilled. This is not without problems. One of these is that when a PULL is encountered, the rest of the loop is executed *as far as* the UNTIL, at which point the loop is terminated.

As an example of PULL you might try the following routine. This contains two nested REPEAT ... UNTIL loops. The inner loop can be ended by pressing S.

```
 10 REPEAT
 20 LET X=0
 30 PRINT "OUTER LOOP, PASS";V
 40 REPEAT
 50 LET A$=KEY$:IF A$="S" THEN PRINT "INNER LOOP
    TERMINATED": PULL
 60 LET X=X+1:PRINT TAB(18)"INNER LOOP, PASS ";X
 70 WAIT 50
 80 UNTIL X=10
 90 LET V=V+1
100 UNTIL V=10
110 PRINT "END"
```

As you will see from this example, using PULL can be useful for staying in an inner loop – i.e. making sure that the loop is repeated until the UNTIL condition is met, but also making the loop repeat from its start, after the PULL.

You may also notice that, if you press S when the inner loop PRINTs the message 'INNER LOOP, PASS 9', the Oric will crash your program with a 'BAD UNTIL' error message. The reason for this is that given above: when a PULL is encountered, the Oric will perform all the actions up to the next UNTIL, when it will terminate the REPEAT . . . UNTIL loop. If you PULL a loop when the flag you use to terminate the loop has been set, the Oric may already have terminated the loop itself – i.e. has already dropped the UNTIL from its memory.

There is no simple way round this problem, which belongs firmly in the realm of advanced programming techniques.

Integer variables

When you use variables you normally use statements like:

```
30 LET G=10
```

This sets up space in memory for a variable called G. A variable can hold a number with decimal places.

The Oric can also make use of what are known as integer variables. These variables can only hold whole numbers (no decimal points) and take up less space in memory. If you know that a variable will only ever be a whole number, then you can specify that a variable is an integer variable like this:

```
30 LET G%=10
```

Of course, every time you want to use the variable, you have to do so with the per cent symbol after its name.

In theory, and indeed on most other machines with integer variables, using integers should make your program run slightly faster. On the Oric, however, integer maths slows down number-handling as does using integer variables in FOR . . . NEXT loops. This is a common ruse of programmers and helps to speed up execution of the loop – not so on the Oric. To prove the slower program execution under integer maths, try the two programs which follow and time them.

```
 10 REM "normal" numeric
    variables
 20 FOR A=1 TO 10
 30 FOR C=1 TO 100
 40 LET N=N+1
 50 LET N=N*N
 60 LET N=N/N
 70 NEXT
 80 NEXT
 90 CLS
100 END
```

```
 10 REM integer
    variables
 20 FOR A=1 TO 10
 30 FOR C=1 TO 100
 40 LET N%=N%+1
 50 LET N%=N%*N%
 60 LET N%=N%/N%
 70 NEXT
 80 NEXT
 90 CLS
100 END
```

ELSE

In the program which drew four circles, you will have noticed the use of the word ELSE to extend the use of IF . . . THEN. Its main use is to make sure that when an IF . . . THEN statement 'fails', something will still happen. In this sense it's a fail-safe that you can use to make certain that variables will take on certain values. In English we might say 'IF it's fine tomorrow THEN I'll go for a walk, otherwise I'll read a book'. If you miss out the 'otherwise', then what might happen if it isn't fine is indeterminate. The 'otherwise' makes what will happen if it's not fine much clearer and it's important when you're working with computers to be as precise as possible at all times.

ELSE can be followed by an expression, such as one to change the value of a variable; a PRINT statement; a GOTO or a GOSUB; and so on. There is, unfortunately, a problem with the way ELSE works on the Oric which is discussed in Chapter 15.

Functions

There are a number of operations that the Oric can carry out that take a different form from the commands you have been using up to now. These are called *functions* and they are always used as part of an ordinary instruction, rather than as instructions on their own. Some of the functions carry out mathematical operations, others enable you to process strings, and others provide access to extra facilities of the computer. Every function consists of a function name followed by parentheses containing whatever it is to operate on, called in technical language the function's 'argument'. For instance, one function is called 'Integer Part' and is written INT(X), where X is some number. This quite simply takes the number inside the brackets and discards any fraction following the decimal point, retaining only the whole number or 'integer' part. Thus:

```
PRINT INT(7.91)
```

gets the reply:

```
7
```

This is often useful in programming, for example in rounding off numbers. Also, to find out whether one number is divisible by another, do the division and test to see whether the integer part of the answer is the same as the answer itself. If it is not, then there is a fraction left over and the numbers you are testing are not divisible.

The function SQR finds the square root of its argument:

```
PRINT SQR(25)
5
```

Other functions carry out mathematical tasks such as finding sines, cosines, logarithms and the like. Unless you aim to do much mathematical work with your Oric it is unlikely you will need to use these.

One very important group of functions are those for handling strings. These enable you to take sequences of words apart and process them in elaborate ways. One of these is LEN, for 'length'. This gives you a number (not a string) telling you how many character lengths there are in the string you specify:

```
LET A$="WINSTON CHURCHILL"
LET N=LEN(A$)
PRINT N
```

The computer replies:

```
17
```

Another string function is LEFT$. This has *two* arguments inside the brackets, separated by a comma. It takes the left-hand end of the string named in the first argument and gives you another string consisting of however many characters are specified by the second argument:

```
PRINT LEFT$(A$,7)
```

The machine replies:

```
WINSTON
```

The function RIGHT$ works in a very similar way to LEFT$. RIGHT$ also needs two arguments, the string or string variable to be used and the number of characters to take from the right-hand side of the string. For example:

```
LET R$=RIGHT$("STRONG",2)
```

will put the letters NG into R$. Of course, the second argument must not be negative. If it is larger than the length of the string, the entire string will be taken. RIGHT$ is useful for stripping unwanted characters from the start of strings.

Likewise 'MID$(X$,M,N)' gives you a string N characters long starting from the Mth character of string X$:

```
LET A$="WINSTON CHURCHILL"
PRINT MID$(A$,5,6)
TON CH
```

Try using these functions to write a program that will take any string (someone's name, for instance) and print it out *backwards*. You will need to use the MID$ function to take just one character at a time from the string, starting at the right-hand end and working left. A program to do this is given at the end of the chapter.

As well as taking strings apart, it is possible to put them together, simply by using a plus sign:

```
LET P$="BEE"
LET Q$="BOP"
LET R$=P$+Q$
PRINT R$
```

The machine replies:

```
BEEBOP
```

This is sometimes called 'concatenation'.

Arrays

It often happens in programming that we have to handle a large number of items of data that are separate but closely related in some systematic way. We may want to store in the computer a list of things, or a table of information, and we want to be able to treat this as one thing, rather than as separate variables each with a different letter label (A,B,C,D . . .). A shoe salesman might, for instance, have a list of prices:

Style 1 costs £ 9.50
Style 2 costs £26.45
Style 3 costs £17.95
Style 4 costs £11.75
etc.

To give every item in this list a separate variable name would be a nuisance. What the salesman would like is to be able to refer to each entry by its position on the list: $Price_1$, $Price_2$, $Price_3$, and so on. The small number is known as a 'subscript'. Basic allows you to do this by putting the subscript in parentheses: P(1), P(2), P(3). So the list can be entered into memory like this:

```
LET P(1)=9.5Ø
LET P(2)=26.45
LET P(3)=17.95
.
.
```

Then the command

```
PRINT P(2)
```

gets the answer

```
26.45
```

The clever thing about this is that *a subscript can itself be a variable.* To see how this works, we can write a program to retrieve any required price from memory. The program asks for the style number wanted, looks it up in the list, and prints the answer:

```
1Ø PRINT "WHICH STYLE"
2Ø INPUT S
3Ø PRINT P(S)
```

Look at that carefully to see how it works. If the user is interested in, say, style 3, he or she types 3 in reply to the INPUT request. The program sets S equal to 3, and in line 30 puts 3 inside the brackets, so it prints P(3).

Here is a program to print out all the prices for styles 1 to 10:

```
1Ø FOR S=1 TO 1Ø
2Ø PRINT "STYLE" S " COSTS £" P(S)
3Ø NEXT
```

You might like to write a program to find which style has the highest price. Try drawing a flowchart first.

In Basic a list can easily become a *table*. Suppose each shoe style comes in a

range of sizes, and each size can be a different price. On paper this would be shown as a table, like this:

			Size		
	1	*2*	*3*	*4*	*5*
Style 1	9.50	9.85	10.25	10.45	10.80
2	26.45	26.95	27.50	27.95	28.70
3	17.95	18.30	18.95	...	...
4	11.75	12.15	...	...	...
.	...	...	...	...	...
.	...	...	...	...	...

In the computer this is easily done by having *two* subscripts, separated by a comma: P(2,5). Either or both of these can be variables.

Then suppose that the price also depends upon the *colour* of the shoe. This means adding a third dimension to the table, something that is difficult to do on paper but with the computer all you have to do is add a third subscript: P(2,5,4). You can even have a fourth or fifth dimension. These become a bit mind-blowing because you cannot visualize them – they cannot exist in the physical world but the computer accepts them quite happily. You can have as many dimensions as you like, with the limitation that you very soon run out of memory in the computer.

Lists and tables such as these are known by the overall term of 'arrays'. Strings as well as numbers can be held in arrays. Suppose the shoe salesman wants to include the name of each style in the computer program. He simply adds another list, in this case a 'string array':

```
LET N$(1)="BROGUE"
LET N$(2)="JACK BOOT"
LET N$(3)="BROTHEL CREEPER"
LET N$(4)="TRAINING SHOE"
.
.
```

You might think that each string in a string array would have to be the same length, but this is not the case. You must remember, though, that you cannot mix numerical arrays and string arrays.

DIM

One problem with arrays is that they can take up a great deal of space in memory, depending on how large a range of subscripts you use. The computer has to leave room for whatever subscripts may come along. Normally it assumes that no subscript is going to be larger than 10. If you are going to use subscripts larger than 10, you must tell the machine beforehand, by specifying the dimensions of your array with the instruction DIM. For a two-dimensional array called T with the first subscript going up to 40 and the second up to 15, you say:

```
DIM T(4Ø,15)
```

READ and DATA

An array cannot be written as such as part of your program; any program which uses an array has to *construct* it with normal Basic commands before it does anything else. In the examples so far, each element of an array has been specified with a LET instruction, but it can be tedious to have to write dozens of LET instructions at the beginning of a program. The alternative is to put the items of data in DATA statements, each separated by a comma. The command READ then takes one item of data at a time from the data statements and stores it just like a LET. This part of a program replaces the series of LET commands on page 54.

```
1Ø FOR S=1 TO 2Ø
2Ø READ P(S)
3Ø NEXT
4Ø DATA 9.5Ø,26.45,17.95,11.75
5Ø DATA .........
```

You will find this is the easiest way of filling a list or table that is of any length.

Assignment statements

While the command word LET has been used extensively in this book up to now, it happens that many versions of the Basic language allow it to be omitted. Because no variable name can be the same as a Basic command word, there can never be any ambiguity in writing

```
A=245
```

instead of

```
LET A=245
```

and in fact these two lines produce exactly the same result. The form A=245 or the like is called an 'assignment statement', because it assigns the value 245 to the variable A. You will often encounter it in programs that you read, but you may find it useful to keep on using LET to keep it clear in your mind that every statement is an instruction to the computer.

Spelling a name backwards

```
1Ø INPUT N$
2Ø FOR X=LEN(N$) TO 1 STEP -1
3Ø PRINT MID$ (N$,X,1);
4Ø NEXT
```

5. Even More About Basic

Now would be a good time for you to try writing some more programs on your own. Several suggestions for these are made here; possible solutions for all these are given later in the chapter. You can either look at these solutions right away, or try on your own first and then compare your programs with those given. There is no one right answer for any of these suggested programs – all that matters is that they work! The examples deal with numbers or words, rather than pictures.

It is a good idea to write your programs out on paper first before you try typing them into the computer. You will find that pencil and paper are just that much less inhibiting to your thoughts than the keyboard.

Teacher

This is a program to test someone's arithmetic. In its simplest form, it can take two random whole numbers between 1 and 100 and print them out. The user is asked to type in the sum, and the computer checks to see if the answer is correct. (The computer, of course, never gets it wrong!) If the answer is right, another sum is provided; if it is wrong, the 'pupil' is asked to try again.

The random numbers are generated by the RND function, like this:

```
LET X=INT(RND(1)*1ØØ)+1
```

to give numbers ranging from 1 to 100 (or any other range you choose). The program could also test subtraction, multiplication and division. It could even choose randomly between these four, by getting another random number in the range 1 to 4.

Perpetual calendar

This is an example of using lists, together with READ and DATA statements. The program will tell you the day of the week for any date between the years 1753 to 2199 (Gregorian Calendar). The procedure is as follows:

1. Take the last two digits of the year.
2. Take the number from step 1 and divide by 4, disregarding any remainder.
3. Take the 'key number' of the month from this table, *but* if the year is divisible by 4, let the key numbers for January and February be 0 and 3 respectively, instead of 1 and 4.

January = 1	May = 2	September = 6
February = 4	June = 5	October = 1
March = 4	July = 0	November = 4
April = 0	August = 3	December = 6

4. Take the key number of the century from this table:
   1700s = 4
   1800s = 2
   1900s = 0
   2000s = 6
   2100s = 4

5. Take the day.

6. Add together the numbers from steps 1, 2, 3, 4 and 5 and divide by 7.

7. Take the *remainder* from the division in step 6. This gives the day of the week thus:
   0 = Saturday, 1 = Sunday, 2 = Monday, 3 = Tuesday, 4 = Wednesday, 5 = Thursday, 6 = Friday.

There is plenty of use of the Integer Part function in this program. To divide while disregarding any remainder, you simply take the integer part of the answer. To find the century, divide by 100, take the integer part, and multiply by 100 again. Thus 1865 becomes 18.65, then 18, then 1800. Subtract that from the complete year to get the last two digits: 1865 − 1800 = 65.

To get a remainder, take the integer part of the answer, multiply by the divisor, and subtract from the dividend. Thus 17 ÷ 5 = 3.4, the 3.4 becomes 3, times 5 gives 15, 17 − 15 = 2. As we know already, 17 divided by 5 equals 3, remainder 2.

Dice Game

With this program you can play a two-dice game against the computer. The game is known in the United States as 'craps', and is familiar to anyone who has seen *Guys and Dolls*. You roll the dice, add the numbers together, and if you get 7 or 11 you win. With 2 ('snake eyes'), 3 ('Holy Joe') or 12 ('box cars') you lose. With any other number you roll again, and keep on rolling until you get either 7, in which case you lose, or the number you rolled in the first place, in which case you win.

You simulate the roll of one die by the command

```
LET X=INT(RND(1)*6)+1
```

Do this again for the second die and then add the two scores together.

Prime factors

A prime number is one which is not divisible by anything other than itself and 1. It is useful to be able to divide up a non-prime number into its prime factors, such as 24 = 2 × 2 × 2 × 3. With large numbers this is extremely tedious, but a computer can do it easily, and in fact computers have been used in some important explorations of the theory of numbers.

The procedure is to try dividing the number which you wish to factorize by each prime number in turn: 2, 3, 5, 7, 11, 13, 17 and so on. You keep dividing out the prime factors until they will go in no longer, and then proceed to the

next higher prime number. You can stop as soon as the prime number you are trying is larger than the square root of whatever is left, because there cannot be more than one factor larger than the square root.

There is a trick used in the suggested program for this that is a good example of how computers do things differently from humans. Strictly the machine ought to have a list of all the prime numbers up to however large a number you might want to factorize, but this would be a nuisance to construct and would occupy a lot of space. The program gets around this by dividing by *every odd number* from 3 upwards (there is no point in dividing by even numbers because they are obviously not prime, apart from 2). Of course plenty of odd numbers are not prime, but since the dividing goes upwards from the smallest numbers, all the lower primes will have been divided out by the time an attempt is made to divide in a non-prime number, so that division will always fail. Thus the computer does plenty of futile divisions, but a program which did not do that would have to be much more complicated. The dumb computer does work to save the human programmer effort, and that must be a good thing.

Sorting

Arranging things in order is one of the most useful tasks a computer can carry out. Plenty of sorting programs exist already, but you might like to try writing a very simple one for fun. It should accept a series of numbers from the keyboard and print them out again in ascending order. This will require using a list, into which the numbers are stored as they come in.

You will need to make several passes through the list, first looking for the smallest number, then for the next smallest, and so on. A really elegant program will rearrange the list in memory into the correct order. The simplest way of doing this is to make multiple passes through the list, looking at adjacent entries in turn, and asking, 'Are these two in the correct order?' If not, the program interchanges them. You have to go through the list as many times as there are items in it. This is called a 'bubble sort', because with each pass through, the next largest item comes to the top.

Debugging

Unless you are superhuman, the programs you write will have mistakes in them to start with. There are two kinds of mistakes. First, there are those in which you ask the machine to do something which is incomprehensible or impossible, such as when you misspell a command word. With these the machine will give you an error message, and it should not be too difficult to find out what is wrong.

The most common error message is SYNTAX ERROR, and this is usually caused by misspelling a command word or the omission of a vital punctuation mark. The computer tells you the line number where the error was encountered, and by looking at that line you should be able to spot the error and put it right. Other error messages are caused by more specific mistakes.

In the other sort of mistake, you give the machine instructions that are perfectly valid, but are simply not the right instructions for achieving your desired end. The machine cannot help you here, because it has no understand-

ing of what it is you are trying to do. These errors in programs are often known as 'bugs', and tracking them down can frequently involve detective work that Sherlock Holmes would be proud of.

Some bugs prevent a program from working at all, while others make it give the wrong answers while appearing to work properly, and yet others only cause a problem in particular circumstances. Some big programs have bugs which have lain in them unnoticed for years, until some unusual situation causes them to make the program go haywire. We can only hope that programs that control nuclear power stations or air-traffic control systems contain none of these.

Assuming that you know there is a bug in your program, you first need to satisfy yourself that your original scheme for the program was sound. When you have checked that, you can try a 'hand simulation', going through the program on the screen step by step, pretending to be the computer and doing everything that the computer would. You note down the values of the variables on a piece of paper as you go along. If you have lots of loops, this can take a long time.

The approach to follow after this is to break the program down into sections and try to confirm that each section is working properly. You can do this by putting in extra PRINT statements that will tell you the state of the variables at each stage in the process. You can also interrupt the program before it is finished, either with CTRL C or by extra END or STOP statements inserted. Then you can examine the variables by giving direct PRINT commands. By this process it should be possible to narrow down where the fault might be. It quite often turns out that the problem is not an obvious mistake on your part but a quirk of the Basic language that you were not aware of. Many of these pitfalls are listed in Chapter 13.

When your program is actually working you need to test it to make sure that the answers it gives, or the actions it takes, are the right ones. You need to give it problems for which you already know the answers, to see that it gets them correctly. For instance, you can try out the perpetual calendar with today's date, or the date of the Battle of Waterloo. Try unusual data: very large or very small numbers, and see what happens. Even after all this, you can never be *absolutely* sure your program has no bugs, but you can feel pretty confident!

Sample programs

Teacher

```
10 REM fancy teacher
20 PRINT "HELLO. PLEASE DO THESE SUMS:"
30 PRINT
40 LET A=INT(RND(1)*100)+1
50 LET B=INT(RND(1)*100)+1
60 LET Q=INT(RND(1)*4)+1
70 ON Q GOTO 80,130,180,230
80 PRINT " "A
```

```
 9Ø PRINT "+"B
1ØØ PRINT "-----"
11Ø LET SO=A+B
12Ø GOTO 3ØØ
13Ø PRINT " "A+B
14Ø PRINT "-"A
15Ø PRINT "-----"
16Ø LET SO=B
17Ø GOTO 3ØØ
18Ø PRINT " "A
19Ø PRINT "*"B
2ØØ PRINT "-----"
21Ø LET SO=A*B
22Ø GOTO 3ØØ
23Ø PRINT " "A*B"/"A
24Ø LET SO=B
3ØØ INPUT AN
31Ø IF AN=SO THEN GOTO 34Ø
32Ø PRINT "SORRY, THAT IS WRONG. TRY AGAIN."
33Ø GOTO 7Ø
34Ø PRINT "THAT IS CORRECT. TRY ANOTHER."
35Ø GOTO 4Ø
```

Perpetual calendar

```
 1Ø PRINT "PERPETUAL CALENDAR"
 2Ø PRINT "FINDING DAY OF THE WEEK FOR 1753 TO
    2199"
 25 PRINT
 3Ø DIM K(12)
 4Ø FOR M=1 TO 12
 5Ø READ K(M): NEXT
 6Ø FOR C=1 TO 5
 7Ø READ L(C): NEXT
 8Ø FOR X=Ø TO 6
 9Ø READ D$(X): NEXT
1ØØ INPUT "YEAR, PLEASE"; Y
11Ø INPUT "MONTH (NUMBER)"; M
12Ø INPUT "DAY, PLEASE"; D
13Ø LET C=INT(Y/1ØØ)*1ØØ
14Ø LET YL=Y-C
15Ø LET YM=INT(YL/4)
16Ø IF Y/4<>INT(Y/4) THEN GOTO 18Ø
17Ø LET K(1)=Ø: LET K(2)=3
18Ø LET A=YL+YM+K(M)+L(C/1ØØ-16)+D
19Ø LET X=A-7*INT(A/7)
2ØØ PRINT D$(X)
3ØØ DATA 1,4,4,Ø,2,5,Ø,3,6,1,4,6
```

```
31Ø DATA 4,2,Ø,6,4
32Ø DATA SATURDAY,SUNDAY,MONDAY,TUESDAY
33Ø DATA WEDNESDAY,THURSDAY,FRIDAY
```

This program includes no traps for invalid data, such as someone asking for the 93rd of the month, or for years prior to 1753 (the beginning of the Gregorian Calendar). These traps could easily be added.

Dice game

```
  1Ø CLS: W=Ø: DEF FNR(N)=INT(RND(1)*N)+1
  2Ø PRINT "DICE GAME!"
  3Ø PRINT
  4Ø GOSUB 2ØØØ
  5Ø PRINT
  6Ø CLS
  7Ø INPUT "YOUR BET IS:";B
  8Ø GOSUB 1ØØØ
  9Ø IF SUM=7 OR SUM=11 THEN GOTO 17Ø
 1ØØ IF SUM=2 OR SUM=3 OR SUM=12 THEN GOTO 2ØØ
 11Ø OLDSUM = SUM
 12Ø PRINT: PRINT "ROLL AGAIN": GOSUB 2ØØØ
 13Ø GOSUB 1ØØØ
 14Ø IF SUM=OLDSUM THEN GOTO 17Ø
 15Ø IF SUM=7 THEN GOTO 2ØØ
 16Ø GOTO 12Ø
 17Ø W=W+B
 18Ø PRINT "YOU WIN. YOUR WINNINGS ARE ";W
 19Ø GOTO 40
 2ØØ W=W-B
 21Ø PRINT "SORRY, YOU LOSE. YOUR WINNINGS
     ARE ";W
 22Ø GOTO 4Ø
1ØØØ D1=FNR(6): D2=FNR(6): SUM=D1+D2
1Ø1Ø PRINT D1;"+";D2;"=";SUM
1Ø2Ø RETURN
2ØØØ PRINT "PRESS THE SPACE BAR"
2Ø1Ø GET A$:IF A$<>" " THEN GOTO 2Ø1Ø
2Ø2Ø RETURN
```

This program uses the Oric's facility for user-defined functions. A function is defined in line 10; function R (FNR) is made to 'return' a random number between 1 and N. The function is 'called' in line 1000 to get two numbers (D1 and D2) in the range 1 to 6 for the two dice. DEF FN is a very useful facility and is explained in more detail later.

Prime factors

```
1Ø PRINT "PRIME FACTORS"
15 PRINT
2Ø LET D=3
```

```
 3Ø PRINT "ENTER NUMBER TO BE FACTORIZED"
 35 INPUT M
 4Ø PRINT "HAS THE FACTORS: "
 45 LET N=M
 5Ø IF N/2<>INT(N/2) THEN GOTO 7Ø
 6Ø PRINT 2: LET N=N/2: GOTO 5Ø
 7Ø IF N/D<>INT(N/D) THEN GOTO 9Ø
 8Ø PRINT D: LET N=N/D: GOTO 7Ø
 9Ø LET D=D+2: IF D<=SQR(N) THEN GOTO 7Ø
1ØØ IF M=N THEN GOTO 12Ø
11Ø IF N<>1 THEN PRINT N
115 GOTO 15
12Ø PRINT "NONE. IT IS PRIME.": GOTO 15
```

Again, traps could be added to this program to look for negative or fractional input. It will not work for two or three or for numbers that are too large to be held exactly in the computer. It often happens in computing that a scheme that is basically sound runs into problems with data at the extremes of the possible range.

Sorting

At some time or another you will want to use a 'sort routine' in a program. You might want to sort a list of names into ascending alphabetical order, or a list of numbers into descending order prior to some statistical calculation. If you consult a book on programming you will probably find a flowchart or listing of the bubble-sort algorithm. Oddly, the bubble sort is one of the least efficient and hence slowest sorting methods that you can use. True, it is easy to code and understand, but there are a number of vastly more sophisticated sorting routines which don't occupy too much more code. One such is the 'Shell–Metzner' sort. The next program allows you to compare the efficiency of the Bubble vs the Shell–Metzner sorts. It also demonstrates several of the points made elsewhere in this book.

Line 60 serves to turn off the cursor. This is necessary because of lines 1070 and 2110. These make sure that the cursor returns to the same column on the current PRINT row (column 23) in order to keep the numbers in the same place. Line 170 turns the cursor back on.

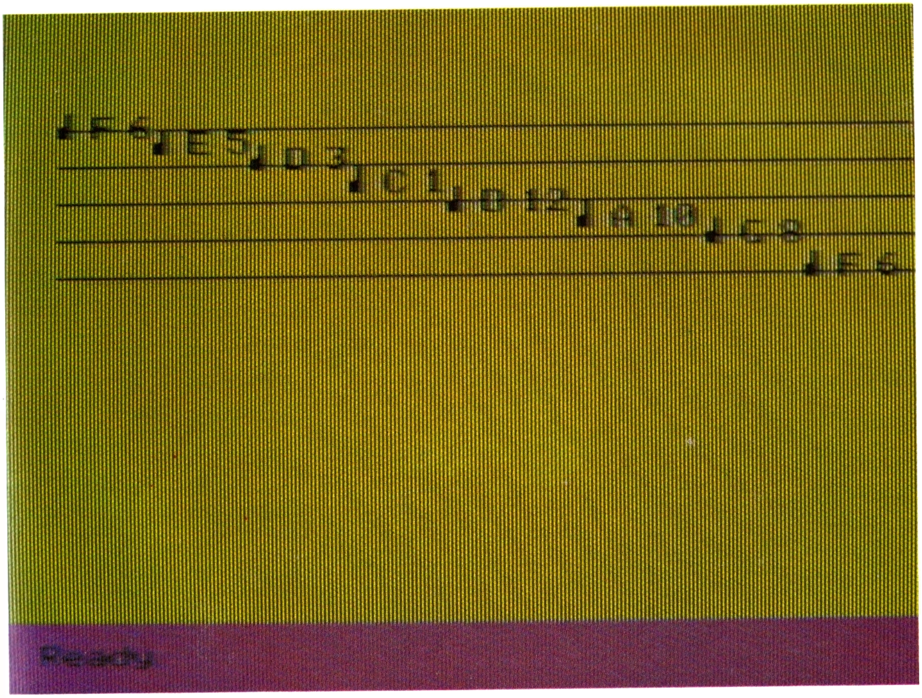
Extensive use is made throughout the program of the Oric's facility for 'computed' GOTOs and GOSUBs, such as 'GOSUB MENU'. This technique makes program development much easier and makes a program far more legible to others. The only thing you have to watch out for is that you don't use other variables whose first two letters are the same as one that you're using for line number references elsewhere in your program.

The program fills an array with 100 numbers chosen at random. You can then elect to have the array sorted via the Bubble or Shell–Metzner routines. You will probably find that the bubble sort takes some 2,500 exchanges to sort the data. The Shell–Metzner sort typically makes only 400 exchanges. Clearly, the Shell–Metzner wins hands down. You might like to change the value of

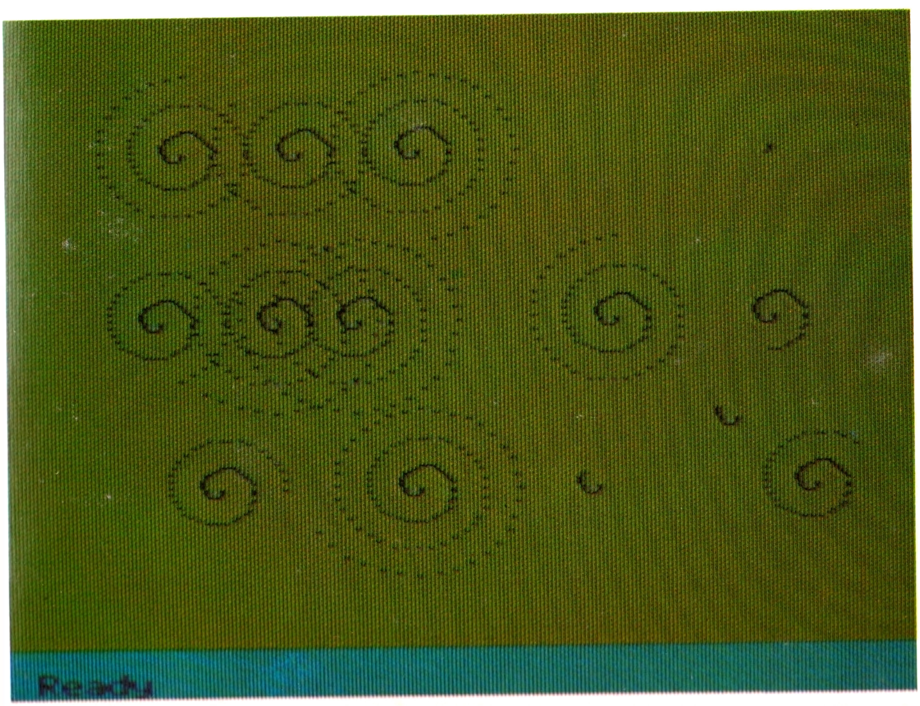
NN in line 50. This dictates the size of the array, i.e. how many numbers are sorted. You will find that the difference between the types of sort increases with larger sets of data. While the bubble sort is quite acceptable for use with small quantities of numbers (<20), there is no point at all using it unless you enjoy making your Oric work unnecessarily hard!

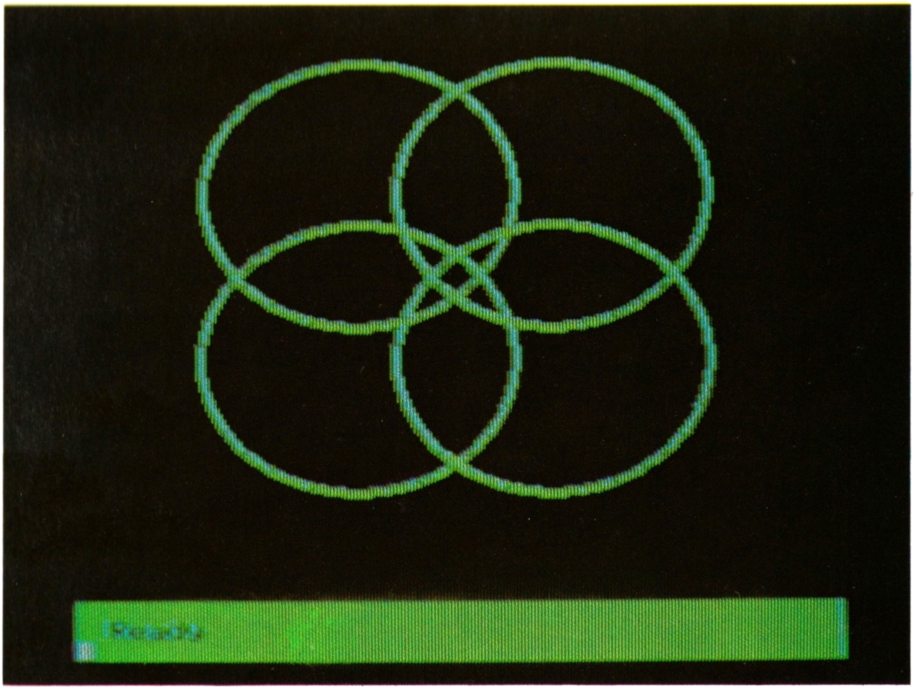
The final point about this program is that when the array is being filled, the user is kept informed that something is going on by the simple expedient of printing a dot after every tenth number (line 3030). This can, of course, be altered to every seventh, twentieth, hundredth or whatever is required.

```
   5 HIMEM#17FF
  1Ø REM comparison & demonstration of
  2Ø REM Bubble & Shell-Metzner sorts
  3Ø REM
  4Ø REM initialize variables, etc.
  5Ø NN=1ØØ: DIM N(NN)
  6Ø POKE 618,PEEK(618) AND NOT 1
  7Ø REM cursor off
  8Ø DEF FNR(X)=INT(RND(1)*1ØØØ+1)
  9Ø MENU=2ØØ: BUBBLE=1ØØØ: SHELL=2ØØØ
 1ØØ NUMFL=3ØØØ: SPACE=4ØØØ: YN=5ØØØ:
     CHECK=6ØØØ
 11Ø CLS: GOSUB NUMFL
 12Ø CLS: C=Ø: GOSUB MENU
 13Ø CLS: ON C GOSUB 1ØØØ,2ØØØ
 14Ø GOSUB SPACE
 15Ø OK=Ø: GOSUB CHECK
 16Ø IF OK=1 THEN GOTO 11Ø
 17Ø POKE 618,PEEK(618) OR 1: CLS: END
 2ØØ PRINT "SELECT ": PRINT SPC(7);"B=BUBBLE/
     S=SHELL-METZNER"
 21Ø C$=KEY$: IF C$="" THEN 21Ø
 22Ø IF C$="B" THEN C=1
 23Ø IF C$="S" THEN C=2
 24Ø IF C=Ø THEN GOTO 21Ø ELSE RETURN
1ØØØ PRINT "BUBBLE SORT": PRINT: PRINT "NUMBER
     OF EXCHANGES=";
1Ø1Ø NE=Ø
1Ø2Ø FOR OL=1 TO NN
1Ø3Ø EX=Ø
1Ø4Ø FOR IL=1 TO NN-OL
1Ø5Ø IF N(IL)<=N(IL+1) THEN 1Ø8Ø
1Ø6Ø T=N(IL): N(IL)=N(IL+1): N(IL+1)=T: EX=1
1Ø7Ø NE=NE+1: POKE #269,23: PRINT NE;
1Ø8Ø NEXT
1Ø9Ø IF EX=Ø THEN 112Ø
11ØØ NEXT
112Ø RETURN
2ØØØ REM Shell-Metzner sort
```



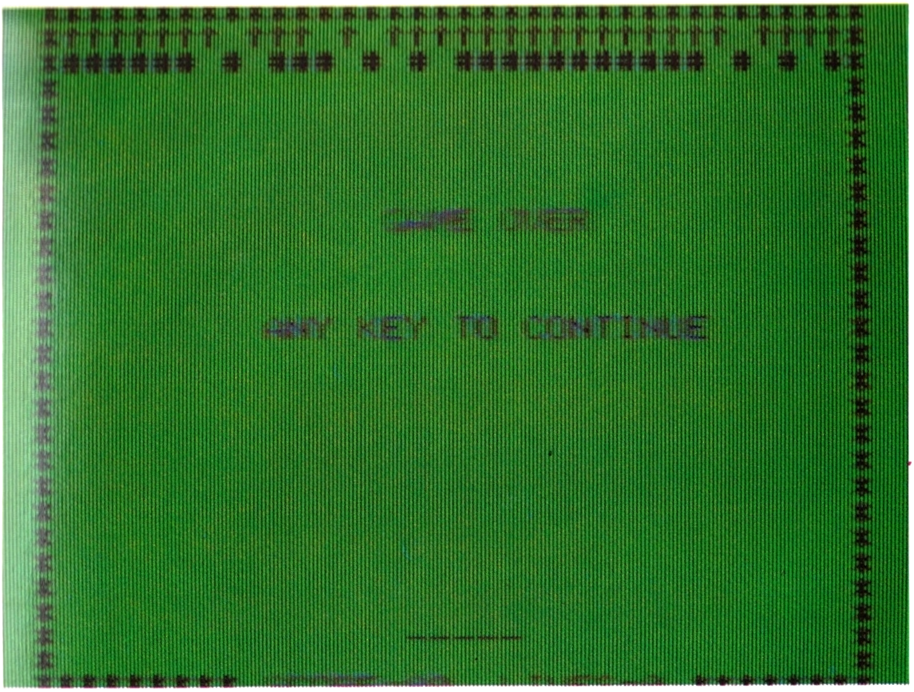
Not only can your Oric play music, it can also show you what it's playing. Here is the scale of C in HIRES graphics



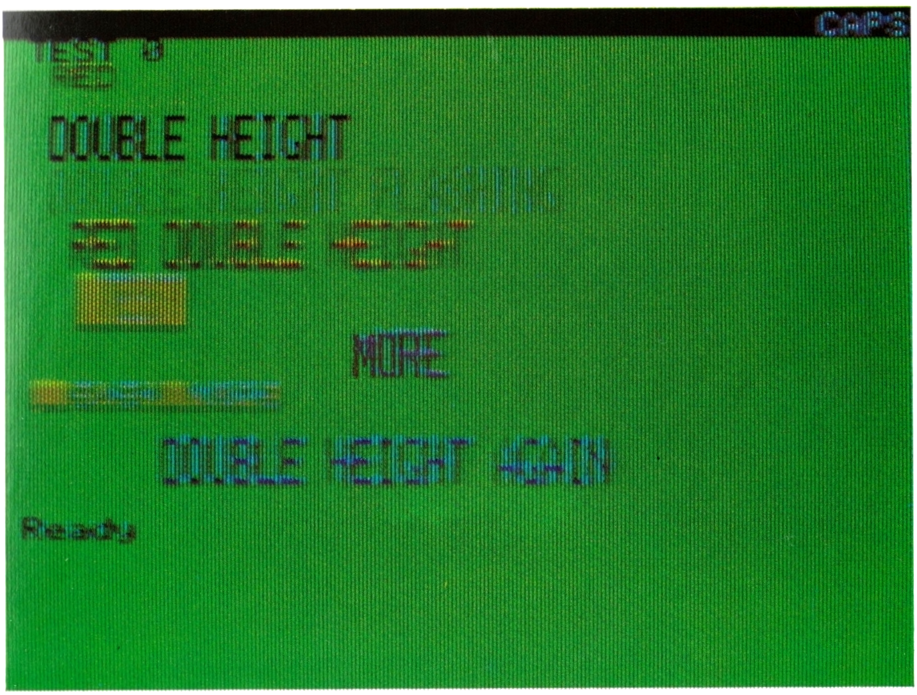


Circles can easily be drawn . . . and erased

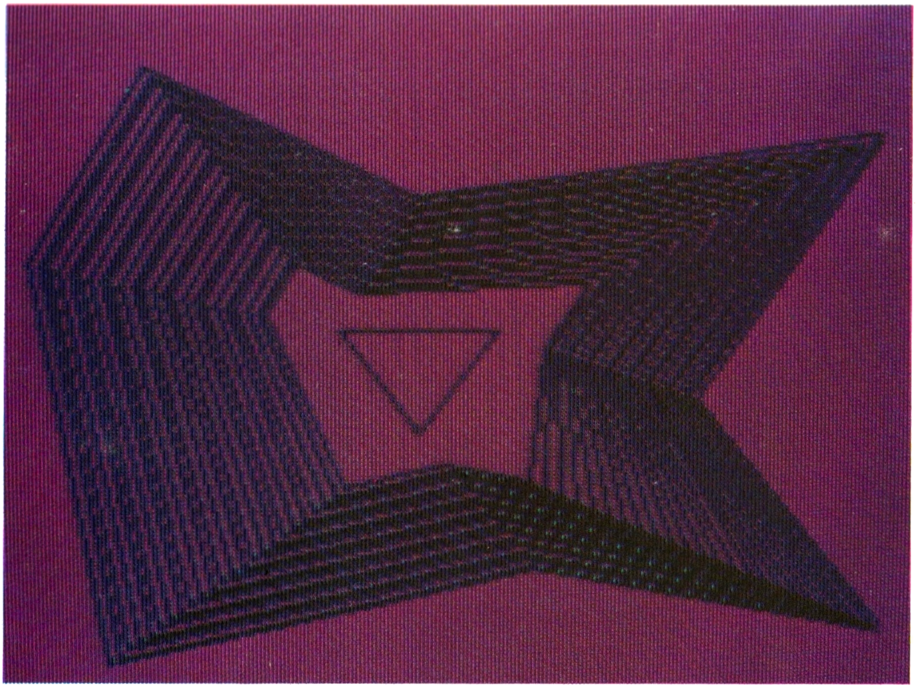
A circular argument? Messages can be written in attractive ways with CHAR on the HIRES screen



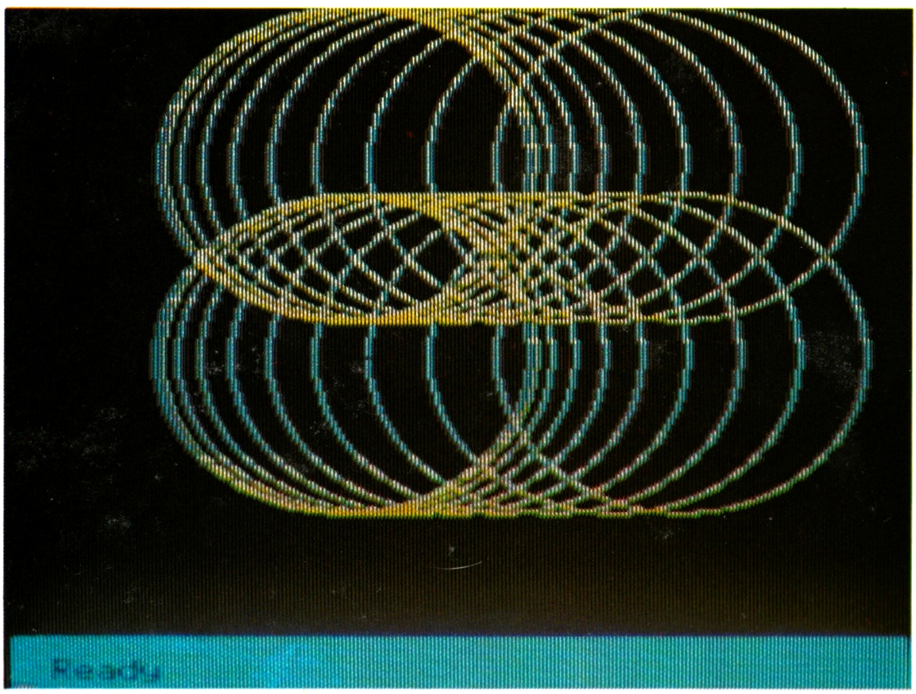
A frame from the finished ball game



Coloured, flashing and double-height letters – some of the effects you can get with serial attributes



A sample of the detailed work your Oric is capable of – 'Contour' lines between two polygons drawn on the HIRES screen



Overlapping shapes can produce new original effects

```
2Ø1Ø PRINT "SHELL-METZNER SORT"
2Ø2Ø PRINT: PRINT "NUMBER OF EXCHANGES";
2Ø3Ø M=NN: NE=Ø
2Ø4Ø M=INT(M/2)
2Ø5Ø IF M=Ø THEN RETURN
2Ø6Ø K=NN-M
2Ø7Ø J=1
2Ø8Ø I=J
2Ø9Ø L=I+M
21ØØ IF N(I)<=N(L) THEN 214Ø
211Ø T=N(I): N(I)=N(L): N(L)=T: NE=NE+1:
     POKE #269, 23: PRINT NE;
212Ø I=I-M
213Ø IF I<1 THEN 214Ø ELSE GOTO 2Ø9Ø
214Ø J=J+1
215Ø IF J>K THEN 2Ø4Ø
216Ø GOTO 2Ø8Ø
3ØØØ PRINT "FILLING ARRAY"
3Ø1Ø PRINT: PRINT "PLEASE WAIT ..."
3Ø2Ø FOR FI=1 TO NN
3Ø3Ø IF FI/1Ø=INT(FI/1Ø) THEN PRINT ".";
3Ø4Ø N(FI)=FNR(1)
3Ø5Ø NEXT
3Ø6Ø RETURN
4ØØØ PRINT: PRINT: PRINT "PRESS SPACE TO
     CONTINUE"
4Ø1Ø C$=KEY$: IF C$<>" " THEN 4Ø1Ø
     ELSE RETURN
5ØØØ C$=KEY$: IF C$="" THEN 5ØØØ
5Ø1Ø IF C$="Y" THEN OK=1
5Ø2Ø IF C$="N" THEN OK=Ø
5Ø3Ø RETURN
6ØØØ CLS: PRINT "CHECK ORDER? ... Y/N"
6Ø1Ø OK=Ø: GOSUB YN
6Ø2Ø IF OK=Ø THEN RETURN
6Ø3Ø CLS: FOR N=1 TO NN: PRINT N(N): NEXT:
     GOSUB SPACE: RETURN
```

This program gives two sorting routines that you could adapt for use with your own programs. The bubble sort is between lines 1000 and 1120, the Shell–Metzner between 2000 and 2160. The numbers to be sorted are in an array N, while NN holds the number of numbers. You can leave out the PRINTs and POKEs if you don't need to see the sort in action; this will have the added advantage of making the routines slightly faster.

Amending data entry

An annoying feature of some programs is that if you make a mistake when entering a long sentence or a large amount of data, you have to type the whole thing in again. This program shows a routine which you can use in your own programs to get round this. It will allow you to change part of a long sentence or a series of numbers with just a few keystrokes. All you have to do is to enter the characters that are wrong, then the replacement characters. You might have entered 'TAKESPOON' in an adventure game, for example. With this routine, all you'd have to do to change this would be to enter 'ES', then 'E S' to put a space between the two words, rather than having to type in the words again and possibly make another error.

The routine is fairly simple as it involves only a few stages to see if the letters to be replaced are in fact in the incorrect sequence; getting the replacement characters; slicing the string into left and right sections and inserting the replacement string.

Of course, there are other checks to be made such as ensuring that the resultant string will not be longer than 255 characters – the longest possible string in Oric's Basic. Another check that has to be made is that the user doesn't attempt to reduce the string to nothing – this may be allowed, but the user must be informed that this has happened and given the option to re-enter it if required.

The program starts by defining A$ to be 'THE RAIN IN SPAIN STAYS MAINLY IN THE PLAIN'. You are then invited to make any changes to this string. This will give you the opportunity to test out the routine to familiarize yourself with its usage. The subroutine you would use in your programs begins at line 1000. Also included is a subroutine which prompts the user to press the space bar in order to proceed (9000 and 9010). This is a very useful subroutine which you will probably find yourself using in most of your programs.

To call the subroutine after an INPUT, simply put the string which might need altering into A$, then execute a GOSUB 1000. When the subroutine RETURNs, A$ will hold the amended string and can be put back into the original string. For example:

```
1ØØ INPUT ANSWER$
11Ø A$=ANSWER$
12Ø GOSUB 1ØØØ
13Ø ANSWER$=A$
14Ø REM rest of program
```

This program shows two oddities in the Oric's handling of strings in INPUT statements: try changing one of the spaces in the sentence. The Oric will not accept a space as a valid input – it makes any string entered as a space empty. Secondly, you cannot just press RETURN when the Oric is waiting for an INPUT – if you do, the question mark is repeated until you make a valid entry. This limitation means that you cannot use the routine as it stands to remove spaces from the sentence. To do this you would have to use the routine described in Chapter 6 to collect characters from the keyboard.

```
  1Ø REM demo of string handling
  2Ø SPACE=9ØØØ: EMPTY=131Ø: START=1ØØØ:
     RET=13ØØ
  3Ø REM replacing substrings
  4Ø A$="THE RAIN IN SPAIN STAYS MAINLY IN
     THE PLAIN"
  5Ø GOSUB START: REM check/alter
  6Ø END
1ØØØ REM subroutine for replacing characters
1ØØ5 CLS
1Ø1Ø LS=LEN(A$)
1Ø2Ø PRINT "CURRENT ENTRY IS:": PRINT A$
1Ø3Ø REM trap empty string
1Ø4Ø IF LS=Ø THEN GOSUB EMPTY
1Ø5Ø PRINT: PRINT "ANY CHANGES? ... Y/N"
1Ø6Ø C$=KEY$: IF C$="" THEN GOTO 1Ø6Ø
1Ø7Ø IF C$="N" THEN GOTO RET
1Ø8Ø IF C$<>"Y" THEN GOTO 1Ø6Ø
1Ø9Ø PRINT: PRINT "ENTER CHARACTERS TO REPLACE:
     ";: INPUT B$
11ØØ LB=LEN(B$)
113Ø TL=1: P=Ø
1132 T$=MID$(A$,TL,LB): IF T$=B$ THEN
     P=TL: GOTO 114Ø
1134 TL=TL+1: IF TL<=LS-LB+1 THEN 1132
114Ø IF P=Ø THEN PRINT "NOT FOUND": GOSUB
     SPACE: GOTO START
115Ø REM
116Ø RO$=RIGHT$(A$,LS-P-LB+1): REM right-hand
     part of main string
117Ø LR=LEN(RO$)
118Ø LF$=LEFT$(A$,P-1): LL=P-1
12ØØ PRINT: PRINT "ENTER REPLACEMENT CHARACTERS:
     ";: INPUT IN$
121Ø LI=LEN(IN$)
123Ø IF LL+LI+LR>255 THEN PRINT "TOO LONG":
     GOSUB SPACE: GOTO START
124Ø REM
125Ø REM
126Ø A$=LF$+IN$+RO$
129Ø GOTO START
13ØØ RETURN
131Ø PRINT "STRING IS NOW EMPTY-OK? ... Y/N"
132Ø C$=KEY$: IF C$="Y" THEN RETURN
133Ø IF C$<>"N" THEN 132Ø
134Ø PRINT "ENTER NEW STRING:": INPUT A$:
     RETURN
9ØØØ PRINT "PRESS SPACE TO CONTINUE"
9Ø1Ø C$=KEY$: IF C$<>CHR$(32) THEN 9Ø1Ø
     ELSE RETURN
```

PEEK and POKE

Together with the commands DEEK and DOKE, these are the Oric words which allow you to alter the contents of RAM (Random-Access Memory) addresses. They can be used to get a number of effects that are not directly supplied by Oric Basic and hence allow you to do more with the computer.

You can find out the contents of an address by PEEKing it. For example, 'X = PEEK(520)' will give the variable X the same value as the current contents of address 520. Address 520 contains a value which relates to the current key being pressed on the keyboard. You could use it to get information from the keyboard and test it rather faster than by using either KEY$ or GET, since KEY$ has to be either compared to other strings or converted to a number, and both these operations take time.

PEEK only looks at a single address (one byte) so it can only return numbers in the range 0–255. The Oric uses two adjacent addresses to hold some information about itself. To access such two-byte addresses you use the command DEEK (for Double PEEK). For example, the current value of the Oric's internal clock can be found by DEEK(630), since addresses 630 and 631 are used to store this value. The clock counts down from 65535 when the Oric is switched on; when it gets to zero, it starts over again. You could use this information to time portions of your programs, e.g. set a time limit to some game or provide a real-time clock while a program is running. To do this you subtract the value of the clock (PEEK(630)) from 65535 (because the clock counts down, not up), then divide this by 47. This gives seconds. You may have to adjust this latter figure for greater accuracy.

POKE is the complement to PEEK. It allows you to change the contents of a RAM address. Because POKE (like PEEK) can only access a single byte, you can only store numbers in the range 0 to 255 in an address. This command is useful for doing things like changing the INK or COLOUR values of various points on the screen, putting characters and patterns on the screen, changing the speed of the Oric or setting the line width for a printer.

DOKE is the double-byte counterpart of POKE. It allows you to put larger values (0 to 65535) into two successive bytes. This is often used for resetting the clock, changing the size of the text screen being used (to 'protect' areas from scrolling) and so on.

Try 'POKE 48000,100'. You should see the letter 'd' appear at the top left of the screen. This is because 48000 is the address of the first cell (byte) of the screen and 100 is the ASCII code for lower-case 'd'. If you change the 100 to 65, you'll get an 'A'. If you POKE values less than 32 to the screen, you'll affect the INK or PAPER colours of all cells on that line to the right of the cell. The text screen runs from address 48000 to 49119.

You can use DOKE to reset the system clock. You must remember that it counts *down*, and that if it gets to zero, it will reset itself to 65535.

You cannot use POKE or DOKE to affect the contents of the Oric ROM which is to be found between addresses C000 (49152 decimal) and FFFF (65535). This area is reserved for the Oric's Read-Only Memory. It is here that the Basic interpreter and system functions are stored.

You can find out what values are held between these addresses using PEEK and DEEK, but without a disassembler (which will tell you the 'mean-

ings' of the values stored in the ROM) the information will be of little use.

POKE will not operate properly if the second number (the number to be loaded in the address given) is in hex format. Thus 'POKE 1024,#1B' will not put 27 (decimal) into location 1024. You would have to change the command to 'POKE 1024,27' but *not* 'POKE 1024,VAL("#1B")'. However, the first number may be in hex, as in 'POKE#400,27'. When READing data, 'POKE AD, VAL ("#" +A$)' will not work. POKE cannot always evaluate such expressions after the comma, so you would use 'V=VAL("#" +A$): POKE AD,V'. Curiously, DOKE is not affected by the same problems.

Some system functions

RND

The Oric has a number of built-in functions which can prove very useful to the programmer. One of these is the facility for generating random numbers. These are never, in fact, truly random, but follow a pattern which does repeat, even if the pattern is rather long. The basic form of random number generation on the Oric is RND(1). This gives pseudo-random numbers in the range 0 to 0.99999. These can be converted into the range 0 to 9.9999 by multiplying by 10. To get numbers in the range 1 to 10 you would add 1 to the above, as in 'R = RND(1) * 10 + 1'. To get integers out of this you would have to use 'R = INT(RND(1) * 10 + 1)'.

You can usefully define a random number generator function using DEF FN. 'DEF FNR(X) = INT(RND(1) * X + 1)' will set up a function to which you simply pass the high number; it will then generate random numbers in the range 1 to X, as in '190 D = FNR(100):REM GET A RANDOM NUMBER 1 TO 100'.

To get numbers randomly chosen between one number and another (e.g. between 50 and 100), set H to the higher number, L to the lower and use 'INT (RND(1) * (H − L + 1)) + L'.

The function RND can be given numbers other than 1 in the following brackets. If you use a negative number, e.g. RND(−3), then you'll always get the same number (but see later). (This is not quite true: you get extremely low numbers like 3.2765378218E −4.

If you give RND a zero, then you get the last RND number used.

As mentioned above, the numbers produced by the Oric with the function RND are not truly random. What happens is that the Oric will follow a predefined pattern of pseudo-random numbers when RND is used. However, you can get round this, to some extent, by resetting the number at which the Oric begins the sequence. This number is known as the random number seed. Passing different negative values to RND forces the Oric to use a new seed, and so changes the sequence of the 'random' numbers produced by RND. An easy way to reset the seed to some random value is to make use of the Oric's clock, which is held in addresses 630 and 631. These two addresses form a double-byte counter. The contents of 630 may be used, once negated, as the new seed for RND. This should give a more or less random seed, since you cannot predict the value in the counter if a program has had to wait for the user to press a key, or whatever, at some earlier stage in a program.

A 'reset RND subroutine' might look something like this:

```
1ØØØ R=RND(-PEEK(63Ø)): RETURN
```

FRE

If you're using a lot of strings and variables in a program, and constantly changing their 'contents', then you might find with a large program that the Oric seems to slow down at times. What may be happening is that the Oric is having trouble sorting out what memory space is no longer needed or being used.

To invoke the process referred to as 'garbage collection', you can insert the statement '*variable* = FRE(" ")', as in 'GC = FRE(" ")'.

You can also use FRE to find out how much memory is being used by your program and its variables by 'PRINT FRE(0)'.

The zero in brackets is known as a 'dummy argument' because it doesn't matter what you use; a 9 would have the same effect. Some other commands need dummy arguments – POS is one of these.

POS

'C = POS(0)' will put the current column number of the cursor into the variable C. This could be used in a routine which formats text by checking whether the next word to be PRINTed would be split over two lines. You would add the length of the next word to be PRINTed to POS(0); if this was greater than 40, you could issue a PRINT CHR$(13) to move the cursor to the beginning of the next line before PRINTing the word.

Reading the keyboard

Often when you're writing a program, you'll want to collect a single letter or number from the user via the keyboard. This can be done using INPUT, but this has two disadvantages. One of these is that INPUT will accept more than one character, and the second is that INPUT requires the user to press RETURN after he's typed his response.

Single key entries are useful when you want to get a response like Yes or No using only the letters Y and N. Another useful procedure is a message like 'PRESS THE SPACE BAR TO CONTINUE'. Clearly, both these routines should not require the user to press RETURN as well. The Oric provides two methods to get a single key press from the keyboard, without the user having to press RETURN.

GET

One of these is GET. GET can only be used with a string variable. It is used in the form 'GET A$', which will suspend a program until a key is pressed, then put the character of that key into the string variable A$. It would be used in a routine like this:

```
1ØØØ PRINT "PRESS THE SPACE BAR TO CONTINUE"
1Ø1Ø GET A$: IF A$<>" " THEN GOTO 1Ø1Ø
     ELSE RETURN
```

This will PRINT the message, then wait until the user presses the space bar, ignoring all other key presses (including CTRL C), before RETURNing to the statement after the statement which called the subroutine.

KEY$

GET always waits for a key-press. The other method, using the function, KEY$, does not.

KEY$, again, has to be used with a string variable. GET and KEY$ offer the programmer the chance to escape from the tyranny of the Oric's error message 'REDO FROM START' which is given to a user entering a non-numeric value when the Oric is expecting a number.

KEY$ is often used when you want to collect a key press, without waiting for the user to press RETURN or when you simply want to see whether or not a key is being pressed (this is particularly useful in games).

KEY$ might be used as follows:

```
1ØØØ PRINT "PRESS Y OR N FOR YES OR NO"
1Ø1Ø A$=KEY$: IF A$="" THEN GOTO 1Ø1Ø
1Ø2Ø IF A$<>"Y" AND A$<>"N" THEN 1Ø1Ø
1Ø3Ø RETURN
```

Here, line 1010 looks at the keyboard and if no key is being pressed, it repeats the line. If a key has been pressed, control passes to line 1020. If the key pressed is neither Y nor N, line 1010 is repeated. If the key pressed is either a Y or an N, the subroutine RETURNs.

Numerical accuracy

If you are dealing with very large or very small numbers, you may find that the Oric lapses into what is called 'scientific notation' when it PRINTs numbers.

For example, try this:

```
1Ø FOR I=2 TO Ø.ØØ1 STEP-Ø.1
2Ø PRINT I/1ØØ
3Ø NEXT
```

You will notice that the Oric starts displaying small numbers like 0.02, 0.019, . . . 0.11, then starts showing things like 9.99999999E–03.

All this means is that the numbers are outside of its normal range. This last number is the same as 9.99999999 × 10 to the power −3 or 9.99999999/1000.

A similar thing will happen when the Oric encounters very large numbers.

As a 'number crunching' language, Basic leaves much to be desired. Try this:

```
IF 5=SQR(25) THEN PRINT "TRUE" ELSE PRINT
"FALSE"
```

Sure enough, the Oric seems to think that 5 is not the square root of 25.
To confuse matters a little, try this:

```
10 FOR I=1 TO 10
20 S=I*I
30 PRINT I,S;
40 IF I=SQR(S) THEN PRINT "TRUE" ELSE PRINT
   "FALSE"
50 NEXT
```

Oddly enough, sometimes the Oric gets its sums right, other times it doesn't.

These samples simply show that the Oric (and, indeed, all machines running Basic) cannot deal with very small differences between numbers. Put simply, the Oric may calculate the square root of 25 to be 5.0000000001, which is not the same as 5. In the second test, some of the differences between the squares and their roots were so small that the Oric treated them as the same.

The only way round this Basic deficiency is to add 0.5 and then use the INT command to round off numbers to the nearest integer before using the results. Try this:

```
IF 5=INT(SQR(25)+0.5) THEN PRINT "TRUE" ELSE
PRINT "FALSE"
```

DEF FN

The Oric will allow you to define up to twenty-six of your own functions which are executed far faster than calling them as subroutines. Such functions generally don't do much – applications like random number generation are their most frequent use, but they can be used to speed up a Basic program to good effect. Curiously, few programmers seem to make much use of them. Here are some examples of use.

```
10 DEF FNA(N)=SIN(N/(180/PI))
```

This will return the sine of the angle N, doing the translation of degrees to radians as part of the calculation – this means that you don't have to remember to do the conversion in the code each time you want a sine calculated. The function would be used as in:

```
100 S=FNA(60)
```

Here, S would be given the value of sin 60°.

```
10 DEF FNT(N)=65535-(DEEK(630))/47
```

Function T is here passed a dummy argument, N, in its definition and this will also happen when the function is called. This function will return the value (in seconds) of the timer since it was last set to 65535.

You can also define functions which use variables when called:

```
10 DEF FNX(X)=X*X+Y*Z
```

Functions are limited to simple calculations: they cannot operate on strings, string variables, nor can they perform operations like POKE or PRINT.

6. The Oric's Text Screen

The text screen

The Oric's text screen is twenty-seven lines deep by forty columns wide. You can get things to appear on the text screen by using the commands PRINT, PLOT or POKE. You can control the format of letters and numbers on the screen by defining exactly where each item will appear. You can also control the foreground and background colours of the screen and you can vary such aspects as the height of the characters, their colour, whether they are steady or flashing and so on. These are called attributes. You can also define the size and position of the screen; if you only want 10 lines on the screen you can tell this to your Oric quite easily. If you want, you can even make up your own shapes and letters.

PRINT and PLOT

PRINT is the most usual way of getting something to appear on your television screen. PRINT will display items from the current PRINT position – i.e. where the cursor is. Unless you put a semicolon at the end of a PRINT statement, a combination of what are known as 'carriage return' and 'line feed' will be sent to the display. These have the effect of moving the cursor down one line (line feed) and to the beginning of that line (carriage return). These terms are taken from the days when computers used teletypewriters for visual output, before Visual Display Units (VDUs) were widely available. This combination of carriage return and line feed can be accessed using 'PRINT CHR$(13)', because 13 is the ASCII code for these two controlling factors. ASCII, pronounced 'askey', stands for American Standard Code for Information Interchange and gives a number (255) of standard characters and controlling functions. These will be dealt with later in greater detail, but for the time being, note that CHR$(8) is a backspace; so 'PRINT CHR$(8)' will move the cursor one space to the left. 'PRINT CHR$(65)' produces the letter 'A', CHR$(66) is the letter 'B' and so on.

PLOT allows you to define the column and row at which an item should be displayed. There are twenty-seven lines on your Oric's screen: the first of these is line number 0, the last 26. The columns run from 0 to 39. The following shows how to use PLOT.

```
10 CLS
20 PLOT 5,7,"COLUMN 5,ROW 7"
30 PLOT 18,27,"PRESS SPACE TO CONTINUE"
40 REM ... rest of program.
```

The general form of PLOT is 'PLOT C,R,A$' where C is the column number at which to start; R is the line or row and A$ is the item to be printed. PLOT has a number of restrictions. You cannot PLOT more than one item at a time. This means that while you can write lines like this:

```
100 PRINT "ENTER YOUR NAME";
110 INPUT NM$
120 CLS
130 PRINT " HELLO";NM$;" I HOPE YOU'RE WELL"
```

You *cannot* exchange the PRINT statement in line 130 for PLOT. What you would have to do in this example is to add the messages and the name together first. Using PLOT, you might insert line 125 to do this, then PLOT the resultant string:

```
125 M$="HELLO "+NM$+" I HOPE YOU'RE WELL"
130 PLOT 5,3,M$
```

This is rather long-winded way of getting the result you want, but it has the advantage that you have greater control over the precise format of your screen display.

The last argument passed to PLOT may be a number or variable. This is then taken as the ASCII code of the character to be displayed (or the attribute to be set). Values less than 32 will set attributes; 32 to 127 give standard characters; 128 to 159 yield attributes again, but produce a coloured block as well and 160 to 255 repeat the character set, but in complementary colours ('inverse').

PLOTting numbers

A restriction of PLOT is that you cannot PLOT numbers or variables directly. So, while 'PRINT X' is quite acceptable, 'PLOT 3,2,X' is not. To PLOT numbers or numeric variables you have to convert them to their string representation by using STR$.

The Oric does a funny thing with this function as it not only converts the numeric item to a string, which is what you want, but it also adds a 'control character' to the front of the string. This item is invisible, but will make the string appear in green ink if you PRINT it.

For this reason, when using STR$ it is always a good idea to strip the first character off after the conversion. For example, if you have a number in the variable AG and want to PLOT this at line 12, column 14, the following is needed:

```
250 AG$ = STR$(AG): REM convert value of AG to
    characters
260 AG$ = RIGHT$(AG$,LEN(AG$)-1): REM strip
    leading character
270 PLOT 14,12,AG$: REM display result
```

This is not an ideal solution, since it will remove any leading negative sign; a better method is to use the following in line 260:

```
260 IF LEFT$(AG$,1)=CHR$(2) THEN AG$=RIGHT$(AG$,
    LEN(AG$)-1)
```

Using colours

Your Oric lets you define the background and foreground colours of the screen by using the Basic words PAPER and INK. These do just what you might expect – PAPER changes the background colour and INK the foreground – the colour that letters or numbers are displayed in. Each of these words must be given a number which tells the Oric which colour you want. 0 is black, 7 is white. The table shows the values:

Number	*Colour*
0	Black
1	Red
2	Green
3	Yellow
4	Blue
5	Mauve (Magenta)
6	Light blue (Cyan)
7	White

You use these words and numbers like this:

```
10 PAPER 2: INK 0: REM black INK on green PAPER
```

You will notice that when you change the INK colour all the items on the screen change to the new INK colour. That is, you cannot change the colour of one message on its own using INK. Similarly, you cannot change the background for one item alone using PAPER.

Differences between PRINT and PLOT

You have to consider the text screen slightly differently for PRINT and PLOT. When you're using PRINT, the screen is normally twenty-seven lines by thirty-eight character positions per line. Although there are really forty such positions per line, the first two of these are usually reserved for the PAPER and INK values for each line. This is why when you LIST your program, or are entering it, the lines all start at the third position or column from the left. If you 'toggle' CTRL] (or 'PRINT CHR$(29)'), then these two 'protected' positions become free for PRINT, and starting a new line allows you to PRINT at the very left of the screen. (This also allows you to use the very first screen line for PRINT. This is usually reserved as a 'status' line for messages like 'SAVING' or 'LOADING'.) However, because you are *not* now specifying PAPER or INK values for that line, the text will appear in white INK on black PAPER.

PLOT gives you twenty-seven lines by thirty-nine columns and is unaffected by CTRL]. You cannot access the first screen column with PLOT, but you can PLOT in the second, which means that you will affect the INK colour for the line. This second position in a line is column zero as far as PLOT is concerned.

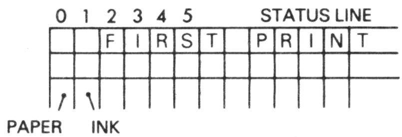
The following programs and diagrams demonstrate these differences between PRINT and PLOT.

```
1Ø PAPER 2: INK Ø: CLS
2Ø FOR RO=Ø TO 9
3Ø PLOT Ø,RO,"USING COLUMN Ø WITH PLOT"
4Ø NEXT
```

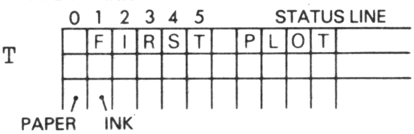
If you now enter 'INK 0' as a direct command, you can see how this overwrites the contents of the second column with the INK attribute and shows clearly that you cannot have an attribute and a character occupying the same place on the screen.

The next few fragments show the differences between PRINT and PLOT.

```
1Ø PAPER 2: INK Ø: CLS
2Ø PRINT "FIRST PRINT
   POSITION"
```




```
1Ø PAPER 2: INK Ø: CLS
2Ø PLOT Ø,Ø,"FIRST PLOT
   POSITION"
```




```
1Ø PAPER 2: INK Ø: CLS
2Ø PRINT CHR$(29)
3Ø PRINT "FIRST TWO COLUMNS NOT PROTECTED"
4Ø WAIT 1ØØ
5Ø PLOT Ø,Ø,"PLOT NOT AFFECTED"
6Ø PRINT CHR$(29)
```

Colours

Although you can't use the commands INK and PAPER to change the foreground and background colours for each PRINT or PLOT, there are ways of achieving just these effects from within BASIC. The ASCII codes give you access to a number of functions to do with screen display. As mentioned earlier, some of the ASCII codes produce letters or numbers when accessed by CHR$. For example, numbers start at ASCII code 48, so 'PRINT CHR$(48)' will result in the number 0 being displayed. Letters start at 65 and end at 90, so 'PLOT 4,5,90' will produce the letter Z at column 4, row 5. These letters are upper case (capitals) and the lower-case letters start at 97 with a and end at 122. There are also various ASCII codes for things like brackets.

Some of the ASCII characters do not produce anything at all when PRINTed, PLOTted or POKEd to the screen. These are referred to as 'control codes' because they affect how *something else* will appear on the screen. You have already seen that CHR$(8) produces a backspace, moving the cursor one space to the left (i.e. erasing any character to the left of the current cursor position). CHR$(12) is what is known as a form feed – this effectively clears the screen by moving everything up by twenty-seven lines. Like CHR$(13) it

was used in the days of teletypewriters to move the paper up a sheet to the beginning of a new page and is still used for the effect today.

Your Oric manual implies that ASCII codes run from 32 to 128. This is not quite true. The full ASCII set starts at 0 and ends at 255. In the Oric's Basic, the numbers from 128 to 255 are duplicates of the codes from 0 to 127. That is, while 'PRINT CHR$(65)' will produce the letter A, so will 'PRINT CHR$(65 + 128)'. However, things aren't quite that straightforward. While 'PRINT CHR$(7)' will produce a ping sound, PRINT 'CHR$(7 + 128)' will not. That is, the symmetry of the two sets of control codes and characters is not perfect.

The best way to find out what each of the ASCII control codes does is to run through the set like this:

```
10 CLS
20 FOR AC=1 TO 255
30 PRINT CHR$(AC);"TESTING CODE";AC
40 WAIT 20
50 NEXT AC
```

When you do this you will get some odd results after some of the codes. This is partly because when the loop PRINTs CHR$(19) the screen goes blank in the sense that nothing PRINTed after this will appear on the screen. This code has the same effect as CTRL S, which effectively disables the VDU (Visual Display Unit).

As with many of the control codes (such as CHR$(6) or CTRL D, which turn the keyclick off), CHR$(19) is a 'toggle' switch. This means that pressing CTRL S will turn the screen on if it's off, or off if it's on. Therefore, to make the testing routine above work properly, you have to RESET your Oric, then add '45 PRINT CHR$(AC)'. This will have the effect of toggling any odd function off, once it's been toggled on by line 30. Some of the control codes will disturb the picture on the TV screen, but no damage will result from this and the TV picture can be set back to normal by pressing the Oric's RESET button.

A list of the control codes and their ASCII codes is given in Fig. 6.1.

Press CTRL and	*Effect*	*PRINT CHR$*
A	Copy text to keyboard buffer	(1)
B	?	(2)
C	STOP program	(3)
D	Toggle PRINT items twice on/off	4
F	Toggle keyclick on/off	6
G	PING – 'Bell'	7
H	Move cursor 1 column left	8
I	Move cursor 1 column right	9
J	Move cursor down 1 row (line feed)	10
K	Move cursor up 1 line (reverse line feed)	11
L	Clear screen (form feed + 'home')	12
M	Carriage return + line feed	13
N	Clear row	14
O	Display off (until CHR$(13))	(15)
P	Toggle printer echo on/off	16

Press CTRL *and*	*Effect*	*PRINT CHR$*
Q	Toggle cursor on/off	17
S	Toggle VDU enable on/off	19
T	Toggle CAPS on/off	20
X	Clear line/keyboard buffer	(24)
Z	?	26
[	Start of ESCAPE sequence (same as ESC key)	27
]	Toggle protected columns on/off	29
Not available on keyboard	Move cursor to top left of screen (home)	30

( ) = Not available

Fig. 6.1 Table of control codes

Most of these commands can be accessed both directly from the keyboard and by using a 'PRINT CHR$' statement in a program. Some only make sense when used directly at the keyboard (e.g. CTRL A for editing), while others are only accessible from PRINT CHR$ statements, e.g. 'PRINT CHR$(30)'.

To get back to colours: there are a number of control characters which lie between ASCII codes 0–26 and 128–154. These can be used to control the colour of individual items on the screen. These codes will also allow you to have messages appear in double-height characters and/or flashing. You can use these to set both the background and foreground for display. The only problem with them is that each one takes up a space before the item you want printed, so you will have to calculate where things will appear with great care.

Fig. 6.2 shows the ASCII codes and their effects.

PLOT CHR$	*PRINT CHR$*	*Effect/Attribute*	ESC*ape and*
0	128	Black ink	@
1	129	Red ink	A
2	130	Green ink	B
3	131	Yellow ink	C
4	132	Blue ink	D
5	133	Magenta ink	E
6	134	Cyan (light blue) ink	F
7	135	White (buff) ink	G
8	136	Single-height steady standard characters	H
9	137	Alternate character set	I
10	138	Double-height characters	J
11	139	Double-height alternate character set	K
12	140	Flashing characters	L
13	141	Alternate character set + flashing	M
14	142	Double-height + flashing normal characters	N
15	143	Double-height + alternate characters + flashing	O
16	144	Black paper	P
17	145	Red paper	Q
18	146	Green paper	R
19	147	Yellow paper	S
20	148	Blue paper	T

PLOT CHR$	*PRINT CHR$*	*Effect/Attribute*	ESC*ape and*
21	149	Magenta paper	U
22	150	Cyan paper	V
23	151	White paper	W
24	152	Text 60 Hz	X
25	153	Text 60 Hz	Y
26	154	Text 50 Hz	Z
27	155	Text 50 Hz	{
28	156	Graphics 60 Hz	|
29	157	Graphics 60 Hz	}
30	158	Graphics 50 Hz	~
31	159	Graphics 50 Hz	←

Fig. 6.2 Table of ASCII character attribute codes for PRINT and PLOT

To use the codes, all you have to do is to add them to the beginning of the item you want displayed. You can do this in a number of ways. If you are PRINTing the item, you can 'send' the ASCII code immediately before the thing you want displayed, separating the two with a semicolon (;) or a plus(+). For example, if you wanted to PRINT the message 'COPYRIGHT 1984' in flashing letters, you would send CHR$(140) before the message, as in:

```
1ØØØ PRINT CHR$(14Ø);"COPYRIGHT 1984"
```

or,

```
1ØØØ PRINT CHR$(14Ø)+"COPYRIGHT 1984"
```

You can even leave out a separator, but this makes programs very difficult to read and debug and is not recommended.

If you're using PLOT then you would have to use column 1 of Fig. 6.2, making the 'control' character CHR$(12), not CHR$(140). 12 is 140 − 128: PRINT and PLOT work slightly differently with control codes. To use the above example with PLOT:

```
1ØØØ CR$=CHR$(12)+"COPYRIGHT 1984"
1Ø1Ø PLOT 3,25,CR$
```

You will remember that you cannot PLOT items together – they have to be joined before using PLOT.

Of course, since you're sending an attribute to the screen with either CHR$(12) or CHR$(140), that attribute will continue to affect items to the right on that row, so it's a good idea to send another control code after the message to make sure that the attribute is turned off. This is only necessary when other items will be PRINTed or PLOTted on the same row, to the right of the special message. To stop the flashing character attribute, you would look up the value in the table – CHR$(136) for PRINT; CHR$(8) for PLOT, then either send the control code after the message, or add it to the message before sending it.

This might mean using program lines like this:

```
1ØØØ PRINT CHR$(14Ø);: REM flash on
1Ø1Ø PRINT "WELCOME TO MY PROGRAM";
1Ø2Ø PRINT CHR$(136): REM flash off
```

Alternatively, you might add all the various bits together, then PLOT the whole thing as in:

```
1ØØØ M$=CHR$(12)+"WELCOME"+CHR$(8): REM flash
     on+message+flash off
1Ø1Ø PLOT 1Ø,12,M$
```

Using PRINT like PLOT

Your Oric version (1.0) of Basic does not have the facility for using the word PRINT to make an item appear at a specified place on the screen. For this you have to use the word PLOT. However, there are ways of getting round this.

Locations 616 and 617 (#0268 and #0269) hold the row and column values for the current PRINT position. This means that if you want to move the cursor ten columns from the left margin before printing something, simply 'POKE 617,10: PRINT "10th Column" '.

Life is not quite so easy with location 616, however. If you want something to be PRINTed on the twelfth line, you have to use lines like this:

```
1ØØ POKE 616,12: PRINT
11Ø PRINT "12th Line"
```

The difference is that you MUST execute a 'dummy' PRINT statement immediately after POKEing location 617 with the required line number before PRINTing the next message.

If you need to combine these two ways of moving the cursor to a predetermined point before PRINTing messages, then you must do so in this order:

1. POKE location 616 with line number.
2. PRINT
3. POKE location 617 with column number.
4. PRINT message.

Of course, the easiest way to use these combinations is to put them in a subroutine to which you pass the two numbers RO and CL for ROw and CoLumn, as in:

```
9ØØØ REM move cursor to CL,RO
9Ø1Ø POKE 616,RO: PRINT: POKE 617,CL: RETURN
```

You would call this subroutine if you wanted 'PLEASE WAIT . . .' to appear at column 14 of row 12, like this:

```
2ØØ RO=12: CL=14;: GOSUB 9ØØØ:PRINT
    "PLEASE WAIT";
```

If you use this subroutine, you must be careful to make sure that you specify *both* the row and column before calling it, or you'll get some messages overwriting previous ones. The other thing you must watch out for is that you don't use the variables RO and CL in any other part of your program. If you do, they may well get set to some illegal values (e.g. negative, or greater than the screen limits) which will cause some unwanted effects. One way to avoid this is to add in a check line at 9005:

```
9ØØ5 IF (RO<Ø) OR (CL<Ø) OR (RO>26) OR (CL>39)
     THEN RETURN
```

What this line does is to test for negative or too-high values in RO and CL. If these are found, it simply RETURNs control back to the statement after the one which called it, i.e. it does not allow illegal values to be POKEd into locations 616 or 617. Of course this may mean that the next PRINTed message will overwrite any previous one, but this is better than an error message or system crash.

Another way to move the cursor to any location on the screen is to DOKE address #12 (18 decimal) with a number between 48000 and 49119. These numbers are the memory addresses of the top-left and bottom-right-hand cells of the TEXT screen itself. Addresses #12 and #13 contain the 'next PRINT address' which is used by the Oric's ROM to work out where in the TEXT memory it should begin PRINTing items. Given that there are forty character positions (columns) per line, and that the screen start address is 48000, it is a fairly simple calculation to work out a number derived from the desired row and column of your message to DOKE into #12. To make the method compatible with PLOT, you have to ignore the first screen line.

The formula for moving the cursor to a given column on a given row is then '48000+((RO+1)×40)+CL−1', where RO stands for the row and CL the column. A program line using this would appear as:

```
1ØØØ RO=1Ø: CL=12: DOKE#12,48ØØØ+((RO+1)*4Ø)+
     CL-1: REM put cursor at col 12 of Row 1Ø
```

Adding 1 to the row is the adjustment for ignoring the top status line, making an RO value of 0 come out as 1.

The following sample program shows the compatibility with PLOT and demonstrates how to use the formula in a subroutine.

```
  1Ø PAPER 2: INK Ø              green PAPER, black INK
  2Ø CLS                         clear screen
  3Ø RO=1Ø: CL=15                set ROw and CoLumn
                                   variables
  4Ø PLOT CL,RO, "PLOTTED"       plot the word 'PLOTTED'
  5Ø WAIT 5Ø                     pause
  6Ø PRINT CHR$(17);             turn off cursor
  7Ø GOSUB 1ØØØ                  move print position
                                   subroutine
  8Ø PRINT "PRINTED";            print 'PRINTED' at new
     CHR$(17)                      position
  9Ø END                         all done
1ØØØ DOKE #12,48ØØØ+((RO+1)*4Ø)+CL-1:RETURN
```

Notice that you have to turn off the cursor by PRINTing CHR$(17), which is the same as CTRL Q, before and after PRINTing your message (lines

60 and 80). This is because, as you may remember, after a PRINT without a semicolon at its end, a carriage return and line feed are sent to the screen. This would result in a coloured block being printed in the first column on the left of the screen.

When using this method, you should be careful to ensure that RO is never less than zero, nor greater that 26. CL may be negative or greater than 39, but you may get some odd results if it is. You could of course add your own routines to check and limit the values of RO and CL.

More on attribute codes

While you're experimenting with the ASCII control codes you will probably come across the one or two values (CHR$(152) is one) which alter the synchronization of the TV to the mains. In this case your TV will flicker a lot and you'll either have to reset your Oric or PRINT the relevant character, e.g. CHR$(154), to get back to 50 Hz (the other is 60 Hz, for the American market).

The nice thing about these control codes is that you can send several before a message. This allows you to choose to have words or numbers displayed in red, double-height, flashing characters on a yellow background if you want, even though the PAPER for the rest of the screen may be set to green and the INK to white. When using the double-height characters you do have to make sure of two things. The first of these is that the thing to be in double height must be started on an even-numbered line; 0,2,4, etc. The second point is that you must PRINT or PLOT the item *twice* in order to get the bottom half of the double height. If you PRINT or PLOT on an odd-numbered line, with the repeated item underneath (on an even line) you will still get your double-height characters, but with the bottom half on the top! The next program shows how to use control codes.

```
  5 CLS
 1Ø INK Ø: PAPER 2
 2Ø PRINT "TEST Ø"
 3Ø PRINT CHR$(129) "RED"
 4Ø A$ = CHR$(131)+"YELLOW"
 5Ø PRINT A$
 6Ø B$ = CHR$(138)+"DOUBLE HEIGHT"
 7Ø PRINT B$: PRINT B$
 8Ø C$ = CHR$(142)+"DOUBLE HEIGHT FLASHING"
 9Ø PRINT C$: PRINT C$
1ØØ D$ = CHR$(129)+CHR$(138)+"RED DOUBLE
    HEIGHT"
11Ø PRINT D$: PRINT D$
12Ø E$=CHR$(142)+CHR$(134)+CHR$(147)+"TEST"+
    CHR$(146)
13Ø PRINT CHR$(4)+E$+CHR$(4)
```

Note how you can either PRINT items twice or precede the PRINTing of an item by CHR$(4) to toggle double-printing items (line 130). This should be turned off after the item with another CHR$(4). Since this rigmarole involves

three PRINT statements and since CHR$(4) has messy effects (try changing line 130 to 'PRINT E$: PRINT E$'), the simpler method is preferable! The program also shows that you do not have to use the semicolon to separate PRINT items which are to appear next to each other on the same screen line (line 30). That is, statements like 'PRINT "THERE ARE" N "ITEMS" ' is the same as ';PRINT "THERE ARE";N;"ITEMS" '. The latter is better as it makes the line more legible to the programmer and leaves less room for error, even if it uses more memory. Note that you can also use the + symbol instead of the semicolon.

The program also shows how you may need to restore the background colour after changing it for a PRINT item (in line 120 'CHR$(146)' is green PAPER).

You should see from this demonstration how attributes sent to the screen occupy one character space each.

It's worth spending some time working out the best order in which to use long strings of attributes (see line 120). Try changing the order to 'CHR$(147)+CHR$(142)+CHR$(134)' to see why.

Inverse colours

PLOT can be used to get an 'Inverse' effect if you set the high bit of characters displayed. To do this you have to add 128 to the code for each character. For example, to display a single letter, such as G, you must first find its ASCII code. Letting your Oric do this for you, you might use a line like:

```
20 AV=ASC("G")
```

Next, you pass this value to the PLOT command, adding 128 to it:

```
30 PLOT 10,12,AV+128
```

You need to compare this to what happens if you haven't set bit seven, so add '65 PLOT 9+K,13,AV'.

Using PLOT's ability to handle 'Inverse' local colours (PRINT will always turn the high bit off, so cannot be used to the same effect) needs more thought when it comes to PLOTting strings, because each character in turn must be dealt with as above. This is easiest in a FOR ... NEXT loop, as in:

```
10 CLS: INK 0 :PAPER 2        set screen
20 A$="TEST OF INVERSION"     define a string
30 LS=LEN(A$)                 get its length
40 FOR K=1 TO LS              loop through
                                 each character
50 T$=MID$(A$,K,1)            isolate a letter
60 AV=ASC(T$)                 get its ASCII
                                 code
70 AV=AV+128                  and set high bit
80 PLOT 9+K,12,AV             plot it
90 NEXT                       get next letter
```

To speed up the program you can condense lines 50 to 80 into the rather complex formula:

```
8Ø PLOT 9+K,12,ASC(MID$(A$,K,1))+128
```

This means that you can delete lines 50, 60 and 70.

Fig. 6.3 shows how the TEXT screen is laid out in memory. Each byte between 48000 and 49119 represents one character position on the screen. The screen is thus said to be 'memory-mapped'.

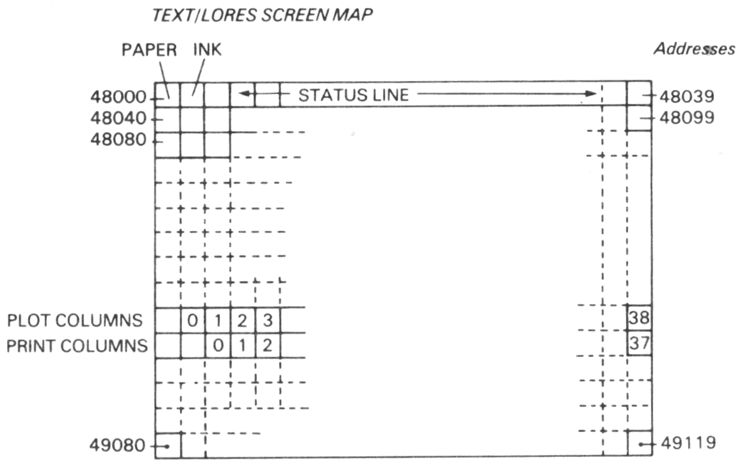


Fig. 6.3 TEXT/LORES screen map

16K owners may subtract 32768 (8000 hex) from these addresses (see Appendix F).

7. More About the Text Screen

Another way of controlling the placing of items on the screen is to implement your own version of PRINT. To do this you can make use of your Oric's facility for having the exclamation mark defined as a command. This lets you extend your Oric's Basic to make the exclamation mark work like a cross between PRINT and PLOT. For example, when you enter and RUN the program given in the manual (Chapter 13, p. 128), you are able to write lines of Basic like this:

```
10 !3,5;"HI THERE"
```

This means the same as 'PLOT 3,5,"HI THERE" '.

The advantage of adding this 'new' word to your Oric's Basic is that you can now PRINT several items together with the exclamation mark, which you cannot do with PLOT. In the example given earlier (showing how you had to add items together before PLOTting them where you wanted) you could rewrite the routine like this:

```
10 CLS
20 !3,0;"PLEASE ENTER YOUR NAME"
30 INPUT NM$
40 CLS
50 !3,0;"HELLO";NM$;". I HOPE YOU'RE WELL."
```

To use the routine to let you do this, first enter the program, being very careful to get the numbers in the DATA statements correct. Then RUN the program, type 'NEW' and you can now enter your own program.

If you want to use the new command ! in any other program, you will have to put the routine as the first few lines of the program, otherwise the Basic will not know what you mean by !.

If you do use the routine, you will have to compensate for the fact that after each use of the exclamation mark to display something, the cursor will be moved down a line. To avoid this simply issue a 'PRINT CHR$(30);' before or after each PRINT statement. CHR$(30) has the effect of putting the cursor at the top-left corner of the screen (home). The semicolon ensures that the cursor stays at (0,0). Missing this, it would be moved to (1,0) due to the carriage return sent after the PRINT statement.

POKEing messages to the screen

To POKE a message to the screen it is probably easiest to write a short routine to handle all the maths. For example, suppose that you wanted to display the message 'PRESS THE KEY OF YOUR CHOICE' starting at column 5 of

line 12 and that you wanted to do so by using a routine that would POKE any message (or control code) to any position specified by the variables RO (for row or line) and CL (for column or character position). The first item to be defined in the routine will be the start address of the screen – 48000. The routine will have to calculate where in memory to put the characters of the string. This is found by adding 40 times the row number plus 1 to 48000 and then adding CL to the answer. Here is just such a routine. To call it from your basic program, simply set A$ to the item you want displayed, put in RO the line or row number and in CL the column number and then execute a GOSUB 9000. The rest will be done for you.

```
9000 CB=48000
9010 LP=CB+(RO+1)*40+CL
9020 FOR K=1 TO LEN(A$)
9030 POKE LP,ASC(MID$(A$,K,1))
9040 LP=LP+1
9050 NEXT K
9060 RETURN
```

The reason for adding 1 to the row in line 9010 is that while the text screen begins at address 48000, the very top line is normally reserved as a status line for such messages as 'CAPS ON'. This means that the first PRINT line really starts at location 48040: 48000 + 40 (forty being the number of characters per line). If ever you use the very top line by POKEing the ASCII codes of characters to it, and if you want to clear the top line, you will have to use a routine like this:

```
1000 FOR K=0 TO 39
1010 POKE 48000+K,32
1030 NEXT
```

The reason for the number 32 in line 1010 is that 32 is the ASCII code for a space – i.e. a blank.

Remember not to set CL to 0 in the routine to POKE a message to the screen because if you do your message will alter the PAPER colour for that line. Similarly if you set CL to 1, you will alter the INK value which is held in the second column.

The text screen in more detail

The Oric has a number of useful tricks available to programmers. Before going into these, you need to know a bit about how the screen memory is mapped out. In the Oric, the screen 'begins' at location 48000 (decimal). Although the screen appears to you as rows and columns, in the Oric's memory the screen begins at one address and ends at another. Because each line is forty characters wide (0 to 39), the second line on the screen begins at location 48000 + 40. To prove this to yourself try this:

```
 5 CLS
10 POKE 48000,65
20 POKE 48040,90
```

This puts the letter A at the very top left of the screen, with the letter Z beneath it. The numbers 65 and 90 are of course the ASCII codes for the letters A and Z respectively. You should see now that as well as using PRINT and PLOT to put items on the screen you can also use POKE. POKE requires rather more thought, however, as you have to POKE each character of a message or string into the correct location in turn, so it's a slower method than PRINT or PLOT. None the less, it's a very handy way of, for example, putting attributes on to the screen. It allows you to split the screen, for instance, so that the left side is green and the right yellow; the following routine shows this.

```
10 CLS
20 PAPER 2
30 CLS
40 INK 0
50 FOR K=0 TO 27
60 POKE 48020+(40*K),147
70 NEXT
```

As you have probably worked out, the middle of the top 'usable' line is address 48020. Line 60, by multiplying the counter K by 40, ensures that the number 147 (which is the ASCII control code for yellow PAPER) is POKED to the middle of each line in turn.

Using techniques like this gives you enormous control over the Oric's text screen, but there's more to come!

A window on the text screen

The screen memory begins at location 48000. However, this number is itself held in locations #026D and #026E (621 and 622). This means that you can 'fool' your Oric into 'believing' that the text screen in fact starts at line 10, for instance. This means that when you use CLS, the first ten lines would not be cleared. This allows you to 'protect' part of the screen. This could be handy if you want to have a title on the screen, on the top line, and never have to worry about reprinting it after every CLS in your program. All you have to do is to POKE location #026D (621 decimal) with the address that you want to be used as the start of the screen. Thus, if you wanted to 'protect' the first line from CLS and scrolling (when the screen was full), you could use a line like this:

```
90 DOKE 621,48040
```

Remember, DOKE is used to put numbers larger than 255 into two successive bytes.

If you wanted to protect the first three lines, this would become:

```
DOKE 621,48000+40*3
```

Any changes to 621 and 622 should be complemented by changes to the next address, 623 (#26F). This byte holds the number of lines on screen and thus effectively works out the end address of the screen when commands like CLS are used. If you want to protect the first three lines, you should then also

POKE 623,27–3. If you don't do this as well as DOKEing the start address to 621, you may get some funny effects at times. For example, if you POKE 623 with a number greater than the number of lines on screen (relative to the start address) then items PRINTed at the 'foot' of the screen will not be visible. POKEing 632 with 255, for instance, then printing 1000 numbers will show what may happen:

```
10 POKE 623,255
20 FOR I=1 TO 1000
30 PRINT I
40 NEXT
```

If you're experimenting with this and can't seem to get the Oric to PRINT normally on the screen, try CTRL C then CTRL L, which should break into the program, then 'home' the cursor and clear the screen. (The 'home' of the cursor is the top left of the screen, so provided you haven't set this to some odd figure, the problem should be cured.)

You can also protect the lower lines of the screen from scrolling or being erased by CLS using these locations. POKEing address 623 with 10 'fools' the Oric into only using 10 lines from the screen start address. Anything that's been PRINTed on the eleventh or higher row number will be unaffected by CLS and other commands that might scroll the screen.

Using these techniques, you can define a 'window' on the screen and PRINT inside it without affecting titles or footnotes. The size of a window may be altered during a program to get different effects.

The next program gives a simple demonstration of a screen window.

```
 10 CLS: PLOT 1,0,"TOP LINE"
 20 DOKE 621,48040: REM protect first line
 30 PLOT 1,26,"LAST LINE"
 40 POKE 623,24: REM protect last lines
 50 CLS
 60 FOR K = 1 TO 40
 70 PRINT "TEST NUMBER";K: WAIT 20
 80 NEXT
 90 CLS
100 DOKE 621,48000: POKE 623,27
```

The last line simply sets the screen back to its normal dimensions.

POS

Sometimes you'll find that you need to know at which column the cursor is, on the current print line. To do this you can get the column number into the variable C (for example) by using the POS functions 'C = POS(0)'. Alternatively, you can PEEK location •0269 (617 decimal) e.g. 'C = PEEK (•0269)'. There is no 'built-in' Basic function for finding out which *row* the cursor is on, so you have to resort to PEEKing address •0268 (decimal 616) for this. It is this sort of information that is particularly useful when you're writing games which use the TEXT or LORES screens.

Improved input

It is possible to improve on Basic's INPUT function. INPUT will accept any length of string (up to 255 characters), which can sometimes be a nuisance. In some applications you might want to limit the user's input to a certain number of characters. For example, if the user is entering the date in the form DD/MM/YY (two characters for the day, two for the month and two for the year, numeric and separated by obliques), then you will want to limit data entry to eight characters. This is often done in commercial programs by displaying a prompt, then a 'data-entry field' whose size is shown by the two symbols < and >. For example, you might want to get the following on screen: 'PLEASE ENTER THE DATE <   >'. Here the cursor would initially be at the first position between the carets. If you used INPUT in the program, the user could overwrite the final symbol which is used to show the limit on the length of the input expected. The routine given next is based around the functions KEY$, CHR$(8), CHR$(127) and CHR$(13).

KEY$ looks at the keyboard and puts the letter of any key being pressed into the string variable given. It is used in the form A$ = KEY$. KEY$ does *not* wait for a key to be pressed. CHR$(8) is a backspace, i.e. the left arrow key of the keyboard. CHR$(13) is a carriage return; the RETURN key generates this value. CHR$(127) is the DELete key.

To use the routine, which is given as a subroutine, you must define a prompt string as PM$ e.g. 'PM$ = "ENTER THE DATE" '. You must use the parameter RO for the row or line that you want the message to appear on. CL should be used for the column that you want the prompt to start at.

Finally you must define MX to be the MaXimum length that the INPUT can be, then pass these to the subroutine with a GOSUB 1000.

The routine will print the prompt, the markers for the start and end of the field and will then collect letters entered at the keyboard into WD$. If the DEL key or left arrow key is pressed, the last letter of the string entered so far is removed. When the user presses RETURN, the routine ends and RETURNS with the data entered in WD$. In short, you 'call' the routine by defining variables like this:

```
 5Ø CLS
 6Ø PM$="PLEASE ENTER YOUR NAME": REM define
    prompt
 8Ø MX=4: RO=1Ø: CL=1: REM define max length
    of input, row and column
 9Ø GOSUB 1ØØØ: REM collect WD$ from data entry
    subroutine
1ØØ IF WD$<>"FRED" THEN GOTO 5Ø: REM password
    is FRED
11Ø REM rest of program
```

Here is the subroutine:

```
1ØØØ CB=48ØØØ: L=LEN(PM$): REM screen start &
     length of prompt
1Ø1Ø WD$="": REM nullify WD$
```

```
1Ø2Ø PLOT CL,RO,PM$: REM print prompt
1Ø3Ø RO=RO+1: REM adjust row
1Ø4Ø SP=CB+(RO*4Ø)+CL+L+2: REM start point for
     carets < and >
1Ø5Ø POKE SP,6Ø:POKE SP+MX+1,62: REM field
     delimiters < and >
1Ø6Ø LP=SP+1: REM start point for letters
1Ø7Ø A$=KEY$: IF A$=""THEN GOTO 1Ø7Ø
1Ø8Ø LW=LEN(WD$): AL=ASC(A$)
1Ø9Ø IF AL=8 OR AL=127 THEN BS=-1 ELSE BS=Ø:
     REM del or left arrow?
11ØØ IF BS<Ø AND LW>1 THEN GOSUB 117Ø: GOTO
     1Ø7Ø: REM if valid backspace, do subroutine
111Ø IF BS<Ø AND LW<1 THEN GOTO 1Ø7Ø: REM
     backspace invalid
112Ø IF AL=13 THEN PING: RETURN: REM end of
     routine-remind user
113Ø IF LW>=MX THEN 1Ø7Ø: REM word too long
114Ø IF AL<65 OR AL>91 THEN GOTO 1Ø7Ø: REM not
     a letter-leave out if numbers wanted
115Ø WD$=WD$+A$: POKE LP,AL: LP=LP+1: GOTO 1Ø7Ø:
     REM add letter to word
116Ø REM backspace routine
117Ø POKE LP-1,32: WD$=LEFT$(WD$,LW-1):
     LP=LP-1: RETURN
118Ø REM print a space to rub out last character,
     strip WD$ of last letter and move letter
     position pointer back one space
```

8. Designing a Game on the Text Screen

You can use the text screen quite effectively for games. One of the easiest to do is a version of 'Breakout'. In this game you control a bat at the bottom of the screen and you have to move it to intercept a bouncing ball. The ball moves around the screen, bouncing off the sides and the top of the screen; if you miss it and it hits the bottom, then either you lose a 'life', or the game restarts. You can add details like patterned blocks at the top of the screen such that if the ball hits one of these, the block disappears and you gain extra points. Then, if the ball hits the very top of the screen, you 'win'. You can also add such things as randomly placed blocks which can deflect the ball, making the game harder to play.

Unfortunately, the game will be rather slow, because Basic is a slow language, but a little thought and ingenuity can make up for slow speed.

The first thing to do is to work out how to display a moving ball on the screen. Fig. 8.1 shows how the ball will appear to move.

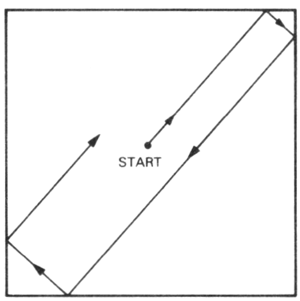


Fig. 8.1 To show how the 'ball' will appear to move around the screen

You must decide where the ball is to start from and what the initial direction will be. Let's start the ball in the middle of the text screen travelling north-east, towards the top right-hand corner of the screen. This would give a player the chance to get ready to move the bat to intercept the ball when it bounces down.

Fig. 8.2 shows in detail how a ball can move. From any square, the ball will move to an adjacent square according to the direction it's going. For each move you have to erase the ball, check that it won't run off the screen and redraw the ball in its new position. To redraw the ball, you will have to calculate the new position with reference to the old position and the direction of travel.

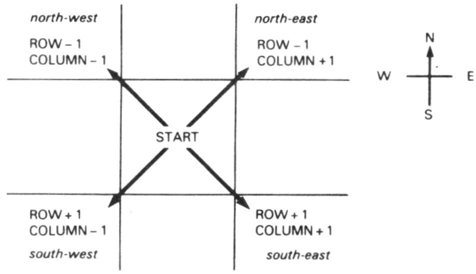


Fig. 8.2 To show how row and column numbers have to be altered to move ball diagonally

As you will be using PLOT to place the ball, you will be using two variables to define the ball's location – BR for Ball Row and BC for Ball Column. How are you going to specify direction? This is quite simple if you think in terms of graph paper. If the ball is moving north-east, the new square will be found by adding 1 to the current column and subtracting 1 from the current row. If the ball is moving in a south-west direction, then the new location can be calculated by subtracting 1 from the column and adding 1 to the row. Fig. 8.2 should make this clear.

This means that you only need two variables to keep track of the direction of travel. One of these will be DR for the direction in terms of rows, the other will be DC for the direction in terms of columns. Each of these variables can only have one of two values – plus or minus 1. So, if you want to make the ball move upwards and to the left, DR will be negative 1 (DR = −1) and so will DC (DC = −1). This makes life surprisingly easy when you come to consider what will happen when the ball hits a wall or the bat. If the ball is moving to the left and it hits a wall, then when it bounces, it will start moving to the right. So, if DC = −1 when a wall is hit, it should change to +1. On the other hand, if DC = +1 when the ball, moving to the right, hits a wall, then DC should change to −1 (DR will behave similarly at the top and bottom).

Put simply, all you have to do is to reverse the sign of the relevant variable for direction (DC or DR) to make the ball appear to bounce off a wall. This means that you have to define in some way how the Oric will 'know' when the ball has hit a wall. This can be done by comparing the current row and column of the ball with limits you set. As you have seen, the lowest row number is 0, the highest 26. The lowest column is 0 and the highest is number 39. *But*, you shouldn't PLOT in column 0 or the colour of the INK will change. This means that we cannot use this column, so you have to set the lowest column you can use to 1.

All in all, you need to set four limits – the highest and lowest column and row numbers that can be used. Let's use HC (highest column), LC (lowest column), HR (highest row) and LR (lowest row). It's always a good idea to make your variables memorable by using the capital letters of what they stand for.

You can now code up these values at the start of the program. These variables are really constants, but Basic has no way of letting you declare constants like other languages. The first few lines will look something like this:

```
1Ø REM declare variables
2Ø HC=38: LC=1: REM high & low columns
3Ø HR=26: LR=Ø: REM high & low rows
4Ø BC=2Ø: BR=12: REM ball starts at column 2Ø
   of row 12
5Ø DC=1: DR=-1: REM starting direction - north-
   east
```

Moving the ball around the screen needs some careful thought as to the order of the operations to be performed. Fig. 8.3 shows a flowchart that makes the necessary sequence clear. You should always draw up a flowchart of the more complex parts of a program *before* trying to write the program in Basic.

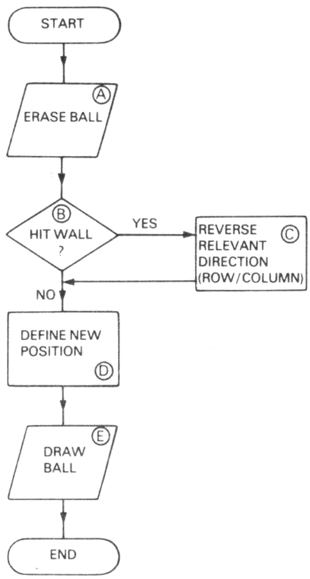


Fig. 8.3 A flowchart showing the 'heart' of the bouncing ball routine

The flowchart (Fig. 8.3) codes almost directly into Basic:

```
     1ØØ REM move ball
Ⓐ    11Ø PLOT BC,BR," ":REM erase ball
ⒷⒸ  12Ø IF BC<LC OR BC>HC THEN DC=-DC:
         REM hit a wall, so reverse column DIR
ⒷⒸ  13Ø IF BR<LR OR BR>HR THEN DR=-DR:
         REM hit top/bottom, so reverse row DIR
Ⓓ    14Ø BC=BC+DC:BR=BR+DR: REM update column and
         row for new position
Ⓔ    15Ø PLOT BC,BR,"Ø": REM draw ball in new
         position
     16Ø REM all done
```

Line 120 checks if the current column of the ball is less than the lowest or greater than the highest columns set in line 20. If either of these limits are

passed, the column direction (DC) is reversed. Line 130 does the same for the ball rows.

You're now ready to test out these two modules – the variable declarations and moving the ball. It's always a good approach to write small sections of program that can be tested out before you make a program more complex. In this way you can make sure that the simpler functions work as they ought to, then you can work out more complex routines to add to a program.

You still need a 'main' part to the program. This is the controlling part which calls up the routines needed to make the whole program work. To test what you've got so far, you can enter a temporary main program at line 1000 like this:

```
1ØØØ REM main program
1Ø1Ø REPEAT
1Ø2Ø GOSUB 1ØØ: REM move ball
1Ø3Ø UNTIL KEY$<>"": REM until a key is
     pressed
1Ø4Ø END
```

Of course you'll have to add '60 GOTO 1000' to jump from the variable declaration section to here and you'll also have to turn the 'move ball' routine into a subroutine by adding '170 RETURN'. The only other thing that's missing is a Clear Screen command. You can put this at line 1005: '1005 CLS'. So far you should have:

```
  1Ø REM declare variables
  2Ø HC=38: LC=1: REM high & low columns
  3Ø HR=26: LR=Ø: REM high & low rows
  4Ø BC=2Ø: BR=12: REM ball starts at column 2Ø
     of row 12
  5Ø DC=1: DR=-1: REM starting direction -
     north-east
  6Ø GOTO 1ØØØ
 1ØØ REM move ball
 11Ø PLOT BC,BR," ": REM erase ball
 12Ø IF BC<LC OR BC>HC THEN DC=-DC: REM hit a
     wall, so reverse column direction
 13Ø IF BR<LR OR BR>HR THEN DR=-DR: REM hit
     top/bottom, so reverse row direction
 14Ø BC=BC+DC: BR=BR+DR: REM update column
     and row for new position
 15Ø PLOT BC,BR,"Ø": REM draw ball in new
     position
 16Ø REM all done
 17Ø RETURN
1ØØØ REM main program
1ØØ5 CLS
1Ø1Ø REPEAT
1Ø2Ø GOSUB 1ØØ: REM move ball
1Ø3Ø UNTIL KEY$<>"": REM until a key is pressed
1Ø4Ø END
```

If you type RUN you'll see your first 'bug' – after the ball 'hits' the top of the screen you get an error message '?ILLEGAL QUANTITY ERROR IN 150'. Why? Try to work out why before you read any further. I have done this deliberately to show you how easy it is to make a very simple but easily overlooked logical error.

To 'debug' the problem you need to know why the program 'crashes' at line 150. One way of doing this is to examine the variables in the line where the program stopped. Try entering 'PRINT BC' and '?BR' (? can be used instead of PRINT to save space). You'll note that the values of these are 33 and −1 respectively. Which has caused the problem? It can't be BC because 33 is a reasonable column number. Since there's no such row or line as −1, the culprit must be BR. Somehow the value of BR has been allowed to get negative. PLOT can't handle negative values, hence the error message.

You may have worked out how this state of affairs has come about. One way of tracing the fault is to insert PRINT statements into the program so that you can follow the exact values of any or all the variables. Since we're only interested in BR, let's insert a 'trace' line at 105: '105 PRINT "BR="; BR'.

If you now RUN the program again, you'll see that BR gets to the value 0 just before the program 'crashes'. What you have to do now is to take this value for BR and work through the 'move-ball' routine, substituting 0 for BR and see what happens:

```
1ØØ REM
11Ø PLOT BC,Ø," "             erase ball – OK
12Ø ...                       BR is not less than LR;BR = 0
                                and LR = 0, so the direction is
                                not reversed
13Ø ...                       not relevant
14Ø BC=BC+DC:  BR= Ø+(-1)     i.e. BR = BR + DR
                              where BR = 0 and DR = −1,
                                so BR is now −1
15Ø PLOT BC,-1,"Ø"            here's the problem!
```

To stop BR ever getting negative like this, there are two solutions (at least). The simplest of these is to alter the value of the constant LR (Lowest Row) to 1 instead of 0. The other method is to alter the IF . . . THEN statement in line 130 to:

```
13Ø IF BR<=LR OR ...
```

i.e. 'If the Ball Row is less than *or equal to* the lowest row'.

You have probably worked out that you will have to make other similar alterations to deal with the other three limits. If you don't believe this, take out the trace line 105, make the changes above, then RUN the program again. It will 'crash' when the ball hits the right-hand wall for the first time.

To make the 'move-ball' routine work properly then, you'll have to rewrite lines 20 and 30 like this:

```
2Ø HC=37:  LC=2 }
3Ø HR=25:  LR=1 } i.e. alter the limits
```

Alternatively, you would alter the checks in lines 120 and 130 to:

```
120 IF BC<=LC OR BC>=HC THEN DC=-DC
130 IF BR<=LR OR BR>=HR THEN DR=-DR
```
i.e. alter the tests

RUN the program now and you should have a ball bouncing round the screen. To stop it just press any key and line 1030 will end the REPEAT loop and the program.

A border round the screen

The next thing to do is to add a border so that the player can see what's going on a bit more clearly.

A border drawn round the edges of the screen presents a number of problems. The worst of these is that the first two columns of the text screen are reserved for the PAPER and INK values for each line. If you use these, then you create problems with the colours on that line.

Clearly, all you have to do is to draw the left-hand border in the third screen column, which means that the right-hand border will have to be in column 37 (PLOT column 36) to make it look symmetrical. It's a pity that you can't draw a border right round the edges of the screen, but that's the way it is with the Oric's unusual way of handling colours.

For the border, you first have to draw the top and bottom, then the left and right sides. Both of these can be done in two FOR . . . NEXT loops. For the top and bottom, you have to start at location 48042. 48000 is the very top left, 48040 is the start of the first line available to PRINT or PLOT, and you have to start in the third column because the first two are reserved for PAPER and INK. This suggests a loop beginning like this:

```
FOR P=48042 TO 48077
```

The second value (the upper limit of the loop) is the thirty-seventh column on the first PRINT/PLOT line which leaves two blank columns to match those left at the left of the screen for the PAPER and INK attributes.

For each P in the loop, you want to POKE some value to the address, (so that a character appears on the screen). The values you use will be the ASCII code of the character you want – if you want stars (asterisks) use the number 42 as in the example. The loop for the top and bottom borders is thus:

```
70 FOR P=48042 TO 48077
72 POKE P,42: REM top
74 POKE P+1040,42: REM bottom: 26*40=1040
76 NEXT
```

These can be combined in the one statement:

```
70 FOR P=48042 TO 48077: POKE P,42: POKE
   P+1040, 42: NEXT
```

A similar process is used for the right-hand and left-hand sides of the border. Here, you want to start at address 48042 as before, but this time the end address (i.e. the upper bound/limit of the loop) will be 48042 + 1040 (1040

is 26 × 40 – there are twenty-six lines of forty character positions) i.e. 49082. Each pass through the loop should deal with a new line so the loop is stepped through in increments of forty:

```
80 FOR P=48042 TO 49082 STEP 40: POKE P,42: POKE
   P+35,42: NEXT
```

The number 35 is added to P in the second POKE for the right-hand side. For this we are using column number 37 to leave two blank columns to the right of the right-hand border to match the two which had to be left at the left (due to the PAPER and INK attribute problem).

Of course you now have to alter the limits that tell your Oric where the walls are, or, when the ball hits them, a star will be erased. The walls were defined in lines 20 and 30 and can now be redefined thus:

```
20 HC=34: LC=4
30 HR=24: LR=2
```

You will also have to call the border-drawing procedure as a subroutine after the CLS in the main program. Amend line 1005 to read:

```
1005 CLS: GOSUB 70: PAPER 2: INK 0
```

Finally, you will have to make the border-drawing routine into a subroutine by adding a RETURN at line 90.

Adding a paddle

The next phase of the development is to work out how to draw a paddle which the player can move around to intercept the ball. You can represent this with five dashes, for example. This can be treated as a string which can be PLOTted. In the variable declaration section at the start of the program you can now add:

```
25 PD$="-----": EP$="     ": REM EP$ is to erase
   paddle - 5 spaces
```

You will also have to add in the starting position for the paddle. If you want it to start in the middle of the screen, subtract 3 from 20 (i.e. half the paddle length from half the screen width) to give the value 17 for the variable PC (Paddle Column). The paddle row will not change. You can declare these in line 26:

```
26 PR=24: PC=17: REM paddle row & column
```

Moving the paddle

You now need a routine to move the paddle, rather like the one devised for moving the ball. The main difference here will be that you will need to check the keyboard to see if the player has pressed a key to move the paddle. The Oric has a function in Basic called KEY$. This simply gets a character from the keyboard whose ASCII number reveals which key is being pressed. The left and right arrow keys generate the numbers 8 and 9 respectively. The whole of

the paddle routine will therefore be based around lines which determine the direction of the paddle according to which key was last pressed at the keyboard. This will look something like the following, assuming PD is the variable which determines the way the paddle is moving (negative 1 for left and positive 1 for right):

```
220 M$=KEY$
230 IF M$=CHR$(8) THEN PD=-1: REM left
240 IF M$=CHR$(9) THEN PD=1: REM right
```

Just as you had to set limits to the ball's movement within the screen, the same must be done for the paddle. Clearly, the leftmost position will be column three of row twenty-four, while the rightmost will be 37 − 5 = 32 (i.e. PLOT column 31). That is, the column the border appears in minus the length of the paddle.

As with the ball, you have to erase the paddle, update its position and redraw it. You must also check if it's to be moved and whether it would go off the screen, or, more accurately, into the left and right borders. A routine to do this is:

```
200 REM paddle routine
210 PLOT PC,PR,EP$: REM 5 spaces to erase paddle
220 M$=KEY$
230 IF M$=CHR$(8) THEN PD=-1
240 IF M$=CHR$(9) THEN PD=1
250 PC=PC+PD: REM change paddle column
260 IF PC<2 THEN PC=2: REM force left limit
270 IF PC>31 THEN PC=31: REM force right limit
280 PLOT PC,24,PD$: REM redraw paddle
290 RETURN
```

The main routine should now be altered so that after calling up the ball-moving process it invokes the paddle-checking and paddle-moving routine:

```
1025 GOSUB 200: REM paddle
```

You now have a ball which bounces round the screen but the game still ends if a key is pressed and the ball ignores the paddle.

The first thing to do, then, is to put a limit to the length of the game. This can be done by altering the condition of the UNTIL statement in line 1030. If you allow the player five lives, then you can use a variable (NL for number of lives) which can be decremented such that when NL = 0, the game is over. So, add:

```
1030 UNTIL NL=0
  55 NL=5: REM number of lives
```

Hitting the ball

What you need next is a way to work out if the ball is about to hit the paddle. This can be done by inserting a simple test into the ball-moving routine, just after the new coordinates have been worked out and before the ball is displayed there. You could work out the precise memory location of the next ball position

using the formula 'NP = 48000 + (40*(BR + 1)) + BC + 1', then PEEK this location to see if the number 45 is stored there (45 is the ASCII code for a minus sign'–'). This is a bit slow and fortunately your Oric provides a built-in function to do all this for you. The function is SCRN, which returns the ASCII code stored in the screen location referenced by C,R where C is the column and R is the row. The lines you need will be:

```
145 AV=SCRN(BC,BR)
146 IF AV=45 THEN DR=-DR: REM hit ball
```

Here, if the paddle is hit by the ball, you simply reverse the direction in terms of rows, i.e. bounce the ball upwards. To add a bit of spice to the game, you could add some sound by putting a PING as part of the 'paddle-bounce' routine:

```
146 IF AV=45 THEN PING: DR=-DR
```

Using AV as the contents of the next ball location allows you to test whether other items have been hit.

Running the program now should give a ball which bounces off the walls and the paddle. All that remains is to add routines for missing the ball, displaying number of lives, a score and so on.

It can often save time to use a variable to act as a 'flag' as to whether some event has occurred. In this case you might use the variable MB (missed ball) which is set to 1 when the ball-moving routine is entered, and is reset to 0 if the player misses the ball. This way, when the subroutine RETURNs control to the main part of the program, you can test MB, calling a subroutine to decrement the number of lives if MB = 1.

```
 1Ø5 MB=1
 165 IF BR>=24 AND AV<>45 THEN MB=Ø
1Ø21 IF MB=Ø THEN GOSUB 4ØØ
```

Line 165 tests to see if the ball row is greater than or equal to 24 (the paddle row), and also checks whether the ball has hit the paddle, if not then MB is set to 0 – the ball has missed the paddle.

New ball

You will now have to enter a rather long section of code for the subroutines to handle a 'new ball' routine, display of the player's status, etc.:

```
4ØØ REM next ball
41Ø REM and lives display
42Ø BR=12: BC=2Ø: GOSUB 6ØØ
425 A$=CHR$(5)+"MISSED!"+CHR$(Ø)
43Ø PLOT 16,1Ø,A$
44Ø WAIT RND(1)*1ØØ+5Ø
45Ø PLOT 16,1Ø,"        ": REM 8 spaces
46Ø NL=NL-1
47Ø NL$=STR$(NL):NL$=RIGHT$(NL$,LEN(NL$)-1)
48Ø B$=CHR$(7)+"LIVES="+NL$+CHR$(Ø)
```

```
49Ø PLOT 22,26,B$
5ØØ RETURN
6ØØ REM pick a random direction
61Ø AN=INT(RND(1)*2): IF AN=Ø THEN DR=1 ELSE
    DR=-1
62Ø AN=INT(RND(1)*2): IF AN=Ø THEN DC=1 ELSE
    DC=-1
63Ø RETURN
```

Explanation of code:

420 Reset the ball row and column, pick a new random direction
425 Define A$ as "MISSED!" in mauve INK (CHR$(5)). CHR$(0) turns off the INK (or the right-hand border would be affected)
430 Display the message (A$)
440 Pause a random period of time
450 Erase message
460 Decrement NL
470 Convert NL to NL$ (without leading character added by STR$)
480 Define B$ as "LIVES=" plus NL$ in white INK
490 Display the message
500 End of subroutine
610 Set DR for row direction of ball at random, according to AN
620 Set DC for column direction at random, according to AN

You should also add '1006 GOSUB 470'. This displays the number of lives left and can be invoked at the start of the game. Note that this involves jumping into the subroutine (starting at 400) several lines through. This can be a very useful trick in Basic, and no other language allows it. It does, however, make your programs extremely difficult to follow.

Bricks

To make the game more interesting, you can now add in routines for placing 'bricks' at the top of the screen, which will be removed if hit by the ball. Each brick hit gives points. You will also have to add routines to check if a brick has been hit and to update the score and display it. The border-drawing subroutine can be amended to:

```
7Ø REM border subroutine
75 UA=ASC("^"): HS=ASC("#"): REM up arrow and
   hash sign
8Ø FOR I=48Ø42 TO 48Ø77
81 POKE I,42
82 POKE I+4Ø,UA: POKE I+8Ø,HS
84 POKE I+1Ø4Ø,42
86 NEXT
9Ø FOR I=48Ø42 TO 49Ø82 STEP4Ø: POKE I,42:
   POKE I+35,42: NEXT
95 RETURN
```

It will put a row of hash symbols and a row of up arrows at the top of the screen.

To check if either of these symbols has been hit, add:

```
155 IF AV=HS THEN SC=SC+5: ZAP: GOSUB 7ØØ:
    DR=-DR: PLOT BC,BR," ": BR=BR+1
16Ø IF AV=UA THEN EXPLODE: SC=SC+1Ø: GOSUB 7ØØ:
    DR=-DR: PLOT BC,BR," ":  BR=BR+2
```

HS was set in line 75 to be the ASCII code for the hash sign, similarly with UA. This is a neat way of not having to look up the ASCII codes; besides, using variables rather than the numbers themselves speeds up the program.

Scoring

Next, add in the following routine which displays the current score. Note that it's only called when the score is altered. If it were called on every go the game would slow down considerably. Because of this, you need to call the subroutine at the start of the game, so add 'GOSUB 700' to line 1006.

```
7ØØ REM score display
71Ø SC$=STR$(SC): L=LEN(SC$): SC$=RIGHT$(SC$,L-1)
72Ø SC$="SCORE="+SC$+CHR$(Ø)
73Ø SC$=CHR$(4)+SC$
74Ø PLOT 8,26,SC$
75Ø RETURN
```

The end

Finally, some message is needed when all five lives have been lost. The final section of code does this, making full use of the Oric's colour, double-height and flashing attributes:

```
1Ø6Ø EG$=CHR$(6)+CHR$(1Ø)+"GAME OVER"+CHR$(8)+
     CHR$(Ø)
1Ø7Ø PLOT 13,7,EG$: PLOT 13,8,EG$
1Ø8Ø AG$=CHR$(14)+CHR$(1)+"ANY KEY TO CONTINUE"+
     CHR$(8)+CHR$(Ø)
1Ø9Ø PLOT 8,11,AG$: PLOT 8,12,AG$
2ØØØ GET A$
2Ø1Ø GOTO 1Ø
```

You now have a game complete in itself. When you have studied its workings you can speed it up in several ways (details are given in Chapter 14). There are several other ways to improve the game: a smaller 'field' would have the effect of speeding up the game. You could have coloured blocks to deflect the ball at random appearing unpredictably, etc.

9. Characters

Characters are the letters, figures and symbols you see on your Oric's keyboard. When you press the key marked 'Q', the letters Q or q may appear on the screen, according to how the shift/caps-lock functions are set. One of the nicer features of your Oric is that you can redefine the character set. This means that you could arrange for the letter B to appear every time you pressed the key marked F, for example. That wouldn't be very useful, but you could reprogram the entire ASCII set of characters, relabel the keys, and turn your Oric into a Russian computer, complete with Cyrillic script. You could design your own symbols if you were doing scientific work or use the facility to create your own aliens, space-rockets, or whatever took your fancy.

Where do characters come from?

When you turn on your Oric, in the few moments delay before the copyright message appears, your Oric is busy loading its characters into RAM. It loads two sets, the normal ASCII set and a teletext set. The normal set comprises 128 characters, each of which consists of dots or cells in an 8 × 6 matrix. The shape of each character is set out using eight numbers – one for each row. For example, the letter E (ASCII code 69) has the configuration shown in Fig. 9.1.

Address	Contents	Binary pattern 7 6 5 4 3 2 1 0 — BITS
46120	62	00111110
46121	32	00100000
46122	32	00100000
46123	62	00111110
46124	32	00100000
46125	32	00100000
46126	62	00111110
46127	0	00000000

Not used

16 4 1
32 8 2

Decimal values of bits 0 to 5

Fig. 9.1 How the letter E is stored in the Oric's memory

Fig. 9.1 shows the RAM addresses whose contents define the shape of the

letter E. Each address contains a value that determines which pixels on a row will be displayed. A zero at a bit position means that that pixel is 'off', i.e. will not be displayed. The contents of the addresses must be greater than 32 or that byte will be treated as an attribute.

The first address of any letter's definition can be found using the formula 'ADDRESS = 46080 + 8*(ASC("*letter*"))'. Alternatively, you can use '46080 + 8*(*ASCII code of letter*)'. (You would need to use the latter to redefine the copyright symbol © (ASCII code 96) since this is not available from the keyboard.)

The shape of any character can be changed by POKEing numbers to the eight successive bytes that define a character. Note that the starting address for the character set changes to #9800 (48K) or #1800 (16K) when HIRES is used. This would alter the formula above if characters were redefined after HIRES.

The reason for using only eight numbers for each character is that your Oric takes the binary representation of each number, to get the bits set (pixels) for each row. The manual explains this quite well. If you imagine each character in its matrix, each point (bit set) that is to be drawn is 1, each blank point 0. To work out what number should appear on each row, you calculate a decimal number from the binary position of each 1 (point that is drawn).

Designing a character

For example, if you wanted to reconfigure the shape of a letter or symbol for use in a game, the first thing you have to do is draw the shape needed on a grid of 8 rows by 6 columns. Then you shade in the cells you want filled in. Fig. 9.2 shows the first stages in designing a stick figure.

ROWS	NOT USED		32	16	8	4	2	1	← DECIMAL VALUES / TOTALS FOR EACH ROW
1					8	4	2		14
2					8	4	2		14
3						4			4
4				16	8	4	2	1	31
5						4			4
6					8		2		1Ø
7					8		2		1Ø
8				16	8		2	1	27

Fig. 9.2 Designing a character

In Fig. 9.2, the numbers at the left are the rows, the numbers at the top and bottom are the decimal values of each column. To get the eight numbers you need for the shape, you add together the values shown at the top or bottom of each column with a point that is set, i.e. shaded. Thus, the first row has columns whose values are 8, 4 and 2 set. Adding these gives 14 as the number for the first row. The second row has the same bits set, the third row only has the value 4, and so on. Altogether you get eight numbers for a shape, in this example they are 14, 14, 4, 31, 4, 10, 10 and 27.

The next thing you need to work out is where in RAM to put these numbers. This will depend on the symbol you want to redefine. It's obviously a good

idea to use shapes that you don't use very often, such as the curly brackets (ASCII codes 123 and 125), the @ sign (ASCII code 64) or the copyright symbol (ASCII code 96, not accessible from the keyboard).

The character set is usually located from address 46080 in RAM and each shape has eight numbers as you have seen. This makes it easy to get your Oric to do all the hard work in sorting out exactly where to put the numbers. If you wanted to redefine the left curly bracket, you might use a routine like this:

```
10 SA=46080+8*ASC("{"): REM starting address
20 FOR I=0 TO 7
30 READ A
40 POKE SA+I,A
50 NEXT: PRINT CHR$(123)
60 DATA 14,14,4,31,4,10,10,27
```

Note how in line 10 you don't even have to look up the ASCII code for the symbol you're reconfiguring; you can just use the Oric's built-in ASC function. After running this program, every time you press SHIFT and the left curly bracket key, or PRINT CHR$(123), you'll get a stick figure, rather than the shape shown on the keyboard. Unfortunately, there's no way to get the 'proper' shape back without either pressing the RESET button, or redefining the character to look as it did before.

An 8 by 6 matrix is a bit restricting in terms of the shapes you can create, but there are various programming techniques you can use to design and use larger shapes. As far as the Oric is concerned, redefined characters are the same as normal shapes; they can be treated like strings, i.e. PRINTed, PLOTted, 'strung together', given different attributes and so on. This means that you could define the copyright symbol to be one alien, then define, say, AL$ like this:

```
AL$=CHR$(96)+CHR$(96)+CHR$(96)
```

Now, PRINTing AL$ will give you three of the shapes together.

Larger characters

Suppose that you wanted to design a long thin shape like a rocket. Putting this in an 8 by 6 matrix would result in a rather squat shape that might be hard to recognize. Why not design the shape in an 8 by 12 matrix, i.e. spread it out over two characters, then add them together before PRINTing them to get the single shape? Fig. 9.3 shows how to approach the problem. First draw up the two grids, then fill in the cells to be set, then add up the binary values, firstly for the left-hand character, which makes the tail, then for the right-hand character making the nose.

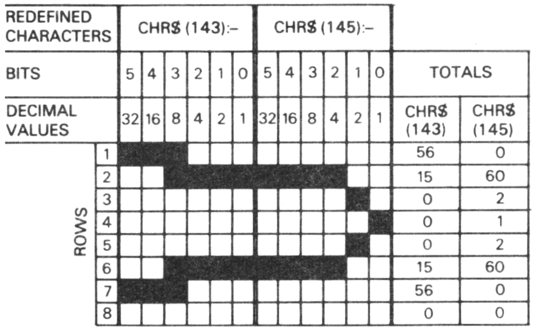


Fig. 9.3 Using two redefined characters to form a large shape

This gives you the following sixteen numbers: 56, 15, 0, 0, 0, 15, 56, 0, 0, 60, 2, 1, 2, 60, 0, 0. All that remains is to use these numbers to redefine two characters, then add these together; the following program does this, using the characters 123 and 125 (the right and left curly brackets):

```
 1Ø LB=46Ø8Ø+(8*123): RB=LB+16: REM addresses
    of left & right characters
 2Ø DATA 56,15,Ø,Ø,Ø,15,56,Ø: REM data for left
    character-tail
 3Ø DATA Ø,6Ø,2,1,2,6Ø,Ø,Ø: REM data for right
    character-nose
 4Ø FOR K=Ø TO 7: REM tail
 5Ø READ A
 6Ø POKE LB+K,A
 7Ø NEXT
 8Ø FOR K=Ø TO 7: REM nose
 9Ø READ A
1ØØ POKE RB+K,A
11Ø NEXT
12Ø R$=CHR$(123)+CHR$(125): REM add tail to nose
    in R$
13Ø CLS: REM initialize (clear) screen
14Ø FOR K=1 TO 25: REM repeat the following 25
    times
15Ø PLOT K,1Ø,R$: REM plot shape
16Ø PLOT K,1Ø,"  ": REM erase it - 2 spaces
17Ø NEXT: REM do it again
18Ø CLS: REM all done
19Ø END
```

This routine is very similar to the one used earlier to define the stick figure. Note that in line 120, the left and right characters are added together, so that PRINTing or PLOTting R$ gets the single shape, line 150.

Even larger characters

Once you realize that you can add ASCII characters together in your redefined shapes, you might also realize that there's no reason that you shouldn't include the non-printing control codes as well. Of course, you can PRINT or PLOT your character definitions with preceding codes for colours of INK, PAPER, etc., but of more interest is the fact that you can also use such codes as 10 (line feed) and 8 (backspace). These two allow you to use four characters to define one shape in a 16 by 12 matrix, increasing the scope of the shapes available to you. To get such large figures, first draw up the four grids as in Fig. 9.4.

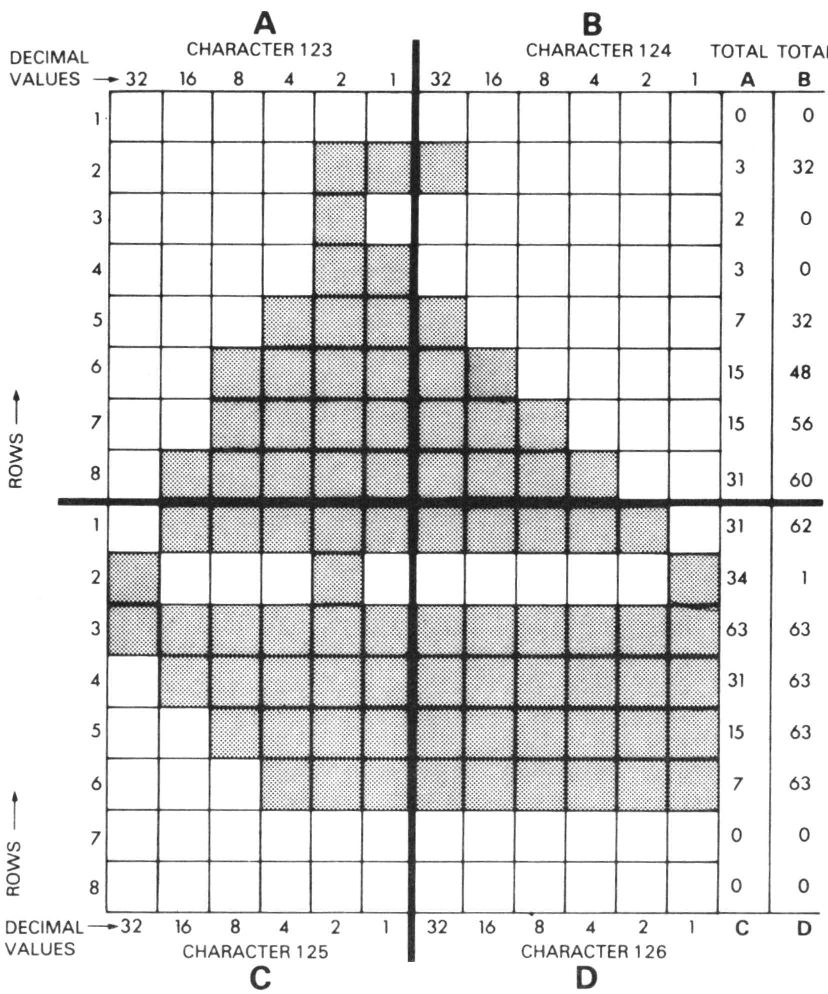


Fig. 9.4 Designing a shape using four characters

Next, you work out the numbers for each character. Then comes POKEing the numbers into RAM to alter each character to your wishes. Finally you have to join up the characters, including the control codes.

This last may need some explanation; to print the four characters in the correct places, you have to work out how the cursor must move. In the figure below, you can see that you can PRINT the first two parts (A and B) as usual, but to PRINT the second two (C and D), the cursor not only has to be moved down a line, but will also have to be moved backwards two spaces.

A	B	B"
C	D	–

The reason the cursor has to be backspaced twice is because after PRINTing part B, the cursor is at a position to the right of B, at the position marked B′. It must therefore be moved down, to the right of D, then back two columns to get to C. All this means is that you define the top two parts as before (in the rocket example) and the same is true for the bottom parts. Then you have to link them with 'CHR$(10) + CHR$(8) + CHR$(8)', i.e. a line feed and two backspaces.

The following routine defines each character, links them, then moves the full shape across the screen. Note that you cannot PLOT a shape defined in this way. This is because the cursor position is *not* affected by PLOT, and for the technique to work it has to be because of the cursor commands. If you do use PLOT, you'll just get the top and bottom parts of your shape on the same line.

```
  5 CLS
 1Ø SA=46Ø8Ø+(123*8): REM start from ASCII
    code 123
 2Ø FOR K=Ø TO 31: READ A: POKE SA+K,A: NEXT:
    REM define characters
 3Ø TP$=CHR$(123)+CHR$(124): REM top
 4Ø J$=CHR$(1Ø)+CHR$(8)+CHR$(8): REM joining
    string
 5Ø BT$=CHR$(125)+CHR$(126): REM bottom
 6Ø SH$=TP$+J$+BT$: REM full figure
 7Ø DATA Ø,3,2,3,7,15,15,31,Ø,32,Ø,Ø,32,48,
    56,6Ø,31,34,63,31,15,7,Ø,Ø,62,1,63,63,63,
    63,Ø,Ø
 8Ø FOR R=1 TO 2Ø
 9Ø PRINT SH$
1ØØ CLS
11Ø POKE #12,48ØØØ+((R+1)*4Ø)+R
12Ø NEXT
```

Another point to note about PRINTing such large figures is that you must be careful not to PRINT them too near the right-hand margin, or they may suffer from 'wrap-around' with half of them appearing at the last position of a line, the rest at the first PRINT position on the next line. Similarly, be careful not to PRINT them too close to the foot of the screen, as you may force a scroll unintentionally, thus affecting your screen display in uncontrolled ways.

Moving characters

Finally in this section, it is quite easy to simulate simple movement of something like a walking stick figure, helicopter and so on by using the character redefinition technique to design two similar shapes, then PRINTing them alternately. The following shows how to achieve this: note lines 120 and 150, which PRINT the alternate characters. Note also lines 140 and 170 which delete the current shape via a backspace; of course, one of these has to have an extra space after it in order to move the cursor along the line. Without this, the figure would simply march on the spot.

```
 1Ø REM define chars 123 & 124
 2Ø DATA 24,24,16,28,26,24,2Ø,22
 3Ø DATA 12,12,24,4Ø,8,24,4Ø,44
 4Ø SA=46Ø8Ø+(123*8)
 45 FOR K=Ø TO 15: READ A: POKE SA+K,A: NEXT
 6Ø REM chars defined
1ØØ CLS: PRINT CHR$(17): REM cursor off
11Ø FOR K=1 TO 2Ø
12Ø PRINT CHR$(123);
13Ø WAIT 2Ø
14Ø PRINT CHR$(8);" ";
15Ø PRINT CHR$(124);
16Ø WAIT 2Ø
17Ø PRINT CHR$(8);
18Ø NEXT
19Ø CLS: PRINT CHR$(17)
2ØØ END
```

The alternate character set

As well as the TEXT and HIRES screens, the Oric has two LORES screens, LORES 0 and LORES 1. These are very similar to each other and each is similar to the TEXT screen! Both LORES screens have 27 rows by 40 columns and are designed to be compatible with such services as Prestel, Oracle and Ceefax. The PAPER colour of these two screens is black – if you try to change it, all you will succeed in doing is putting a two-column band of colour down the left-hand side of the screen. You can change the INK colour of messages in the usual ways with attributes.

You can PRINT, PLOT and POKE in either of the LORES modes, but the default character sets are different. (Default in computer terms means what happens if you don't interfere. For example, when you switch the Oric on, the screen defaults to TEXT mode, the INK default colour is black, the PAPER white.) In LORES 0, the normal character set is used.

In LORES 1, a set of sixty-four graphics characters is available. The characters are very simple, being designed in a matrix of 3 rows by 2 columns. They are loaded into RAM from ROM along with the normal character set when the Oric is turned on. Therefore, they could be redefined, as can the normal set.

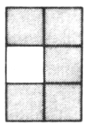
There's absolutely no point doing this, however, because there are only sixty-four possible ways of setting cells in a 3 by 2 matrix, so all the possible characters are in RAM anyway.

To see the alternate character set, try:

```
1Ø LORES 1
2Ø FOR K=32 TO 95
3Ø PRINT CHR$(K);
4Ø NEXT
```

There are other characters available – try changing the 95 to 128. You may also discover the problem that if you fill the screen the Oric jumps back to TEXT mode.

The characters can be selected by using CHR$, but you have to work out the number of the shape you want. First sketch out the 3 by 2 matrix, shading the areas you want displayed. For example:



Next, add the numbers shown in the diagram below which correspond to the areas you have filled in, then add 32.

1	2
4	8
16	32

In the example, this gives 1+2+8+16+32=59+32=91, so the number code of the shape is 91.

That is, to see this block in LORES 1, you would PRINT or PLOT CHR$(91). In LORES 0, however, CHR$(91) would give the left square bracket ([). In any screen mode you can elect to have either the bracket or the graphic block displayed as CHR$(91). In Lores 0, you would precede the CHR$(91) with a CHR$(136), which tells the Oric to use the alternate character set. The same is true in TEXT mode.

In HIRES, you could get the same effect by making the second argument of CHAR a 1 not 0.

You can use CHR$(136) and CHR$(137) to select the normal or alternate character set respectively in either of the LORES modes or in TEXT mode.

The Oric has a bug in its handling of the LORES screens. You cannot scroll the screen at all! If you attempt to do this, the Oric reverts to TEXT mode. You can prove this with the following:

```
1Ø LORES 1
2Ø FOR K=1 TO 3Ø
3Ø PRINT K
4Ø NEXT
```

You will see as soon as the screen is forced to scroll (when the next character would appear off the screen) that when the Oric moves all the screen lines up

one row, the bottom line is in TEXT mode. You could use this effect to get a split screen, for what it's worth, but it should point out the importance of using PLOT to make sure that it doesn't happen.

On some early Orics the teletext characters can only be described as rather lopsided; the following routine should be used at the start of any program which uses the alternate character set:

```
10 FOR A=47368 TO 47858
20 N=PEEK(A)
30 IF N=240 OR N=15 THEN POKE A,-56*(N=240)-7*
   (N=15)
50 NEXT
```

In point of fact, the alternate character set is defined in the usual 8 × 6 matrix in RAM. This means that you can upload the character set definitions between 46080 and 47104 to the alternate character set which lies between 47104 and 47199. This would give you three sets of the normal ASCII characters to play with. Line 30 contains a trick with equals signs to save having two separate IF . . . THEN statements – see page 156 for explanation.

10. The Oric's HIRES Screen

As well as being able to display text, your Oric has what is known as a high-resolution (HIRES) graphics mode. This is invoked by the command HIRES. HIRES mode makes available a number of commands which cannot be used in TEXT or either of the LORES modes. HIRES cannot be used if a GRAB command has been given. You would have to cancel the effects of GRAB by its counterpart RELEASE. GRAB takes some of the area of memory used for displaying the HIRES screen to use for variable and program storage. This is particularly useful on the 16K Oric, where memory may be limited with a large program.

In memory, the HIRES screen runs from address 40960 (A000 in hex) to 48959 (B F3F hex). HIRES mode gives you a screen of 240 columns (each of 6 pixels) by 200 rows, i.e. 248,000 points. You can put the full character set on to the HIRES screen, and you have 240 columns by 200 rows for this. Fig. 10.1 shows how letters etc. are put on the HIRES screen compared with the TEXT screen.

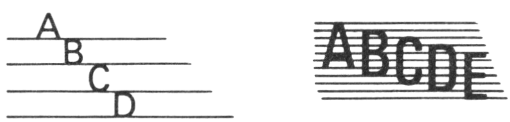


Fig. 10.1 Characters on the TEXT and HIRES screens

The graphics screen is really very like the TEXT screen and you should continue to think in terms of rows and columns. The relationship between the two modes is simple. In TEXT mode, each character is composed in a 6 column by 8 row matrix. As there are 40 columns this gives 240 (40×6) HIRES columns, while HIRES rows are given by $8 \times 25 = 200$.

In HIRES mode there are several commands you can use to put pictures on the screen and, of course, you can POKE numbers into the relevant RAM addresses to access the HIRES screen. Note that many of the commands used in TEXT mode (like PRINT, PLOT, etc.) cannot be used in the same way in HIRES mode.

HIRES leaves three lines blank at the bottom of the displayed screen. These can be used as you would use the TEXT screen. It is rather as if you had defined a small window as described in Chapter 7. Messages can be displayed here using PRINT.

When the Basic interpreter encounters a HIRES command, the screen is cleared, and the cursor put at the top left of the screen. (This also true when

HIRES is left, i.e. when the interpreter finds a TEXT or LORES statement.) To move the cursor around the HIRES screen there are two commands which do this directly and others which will affect the cursor, but whose function is not specifically designed for the purpose.

The two basic functions used to move the cursor around the HIRES screen are CURSET and CURMOV. Both of these require three arguments: a column value, a row value and a cursor value (which sets the cursor to either the background or foreground colour).

The difference between CURSET and CURMOV is that CURSET is 'absolute', that is, it is used to put the cursor at a specified point on the screen. CURMOV, on the other hand, is 'relative'; it is used to move the cursor, say, ten pixels to the right and two pixels down, from wherever it happens to be. You have to be careful when using both these commands that you don't try to put the cursor somewhere off the screen. If this happens, you will get an '?ILLEGAL QUANTITY ERROR' message.

Using these two commands is fairly straightforward;

```
100 HIRES
110 CURSET 120,100,3
```

This fragment enters HIRES mode (line 100), and puts the cursor in the centre of the screen, but not visible. The last argument can only have four values: 0 makes it appear in the background colour; 1 makes it appear in the foreground colour; 2 should invert the cursor colour (but doesn't) and 3 renders it invisible (i.e. does the same as 0).

To move the cursor in the way mentioned above (right 10, down 2), you would use a line like:

```
200 CURMOV 10,2,0
```

This would move the cursor as requested and set its colour to the background. When using these commands, it's often easiest to set up a simple subroutine to which you pass the relevant values in the variables C, R, and FB. This avoids typing errors and problems of forgetfulness. For example:

```
9000 REM move cursor (absolute)
9010 CURSET C,R,FB:RETURN
9020 REM move cursor (relative)
9030 CURMOV C,R,FB:RETURN
```

As will be discussed in the Techniques section, this will slow down your program a bit, but it has the advantage of reducing the amount of memory used and reduces the likelihood of an error.

You could use CURSET to draw lines:

```
10 HIRES
20 INK 0: PAPER 2
30 C=120: R=100: FB=1: REM start in middle of
   screen
40 REPEAT
50 : CURSET C,R,FB: REM put cursor at column
   (C), row (R) in foreground colour
```

```
60 : R=R-1: REM go up a row
70 UNTIL R=50: REM line length=50 (i.e. 100-50)
80 END
```

Fortunately, your Oric provides a much simpler way to draw lines. The command DRAW will draw a line from the current cursor position to the relative coordinates given. As with CURMOV and CURSET, it requires the three arguments C, R and FB (column, row and fore/background). It is important to understand that DRAW is relative (like CURMOV) – that is, it adds the column and row values given to the current cursor coordinates to work out the point to which it should draw the line. DRAW also updates the cursor position, so this may have to be reset in a complex sequence of statements.

To draw a box in the centre of the screen, you could use a sequence like this:

```
10 HIRES: PAPER 2: INK 0
20 CURSET 120,100,3: REM middle of screen
30 CURSET 60,50,1: REM top left of box
40 DRAW 0,100,1: REM down
50 DRAW 120,0,1: REM right
60 DRAW 0,-100,1: REM up
70 DRAW -120,0,1: REM left
80 END
```

PATTERN

PATTERN is an unusual command. It doesn't do anything immediately visible *per se*, but it affects the two commands DRAW and CIRCLE. What it does is to specify how each group of eight bits (a byte) appears. You would use it to get dotted or dashed lines (or a combination of the two). To understand how it works, imagine DRAWing a straight line. This is made up from groups of eight pixels. The table shows how each group is arranged, together with a numerical reference and the binary value for each bit:

7	6	5	4	3	2	1	0	BIT NUMBER
1	1	1	1	1	1	1	1	BASIC SET – SOLID LINE
128	64	32	16	8	4	2	1	DECIMAL VALUES: 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255

When you turn on your Oric, the PATTERN value is set to 255 to give solid lines.

If you want a dotted line, you simply draw up a grid like the one shown, fill in the bits you want set, then add up the binary values given for these bits. This value is then given to PATTERN as in:

```
200 PATTERN 105
```

This example sets the value for lines DRAWn as shown in the table:

7	6	5	4	3	2	1	0	BIT NUMBER
0	1	0	1	0	1	0	1	BITS SET – DOTTED LINE
–	64	–	32	–	8	–	1	DECIMAL VALUES: 64 + 32 + 8 + 1 = 105

A dashed line can be made with PATTERN 231 as shown in the table below:

7	6	5	4	3	2	1	0	BIT NUMBER
1	1	1	0	0	1	1	1	BITS SET – DASHED LINE
128	64	32	–	–	4	2	1	DECIMAL VALUES: 128 + 64 + 32 + 4 + 2 + 1 = 231

CIRCLE

CIRCLE requires two arguments: R (radius) and FB (as with the previous commands). R must be greater than 0 and less than 119. This is because the largest possible circle would have its centre at 120, 100 and its radius would have to be less than 120 to stop the periphery from running off the edges of the screen. The following program draws concentric circles:

```
10 HIRES
20 PAPER 3: INK 0: REM yellow paper, black ink
30 R=1:FB=1: REM radius initial value
40 CURSET 120,100,FB: REM screen centre
50 REPEAT
60 : CIRCLE R,FB: REM draw circle, radius R
70 : R=R+10: REM increase radius by 10
80 UNTIL R=100: REM until radius=100 units
90 END
```

Of course, you can use CURSET to get some quite effective shapes. The following is a general method of producing spirals; you can alter several of the variables to make the spiral take on different characteristics.

```
 10 REM simple spiral
 20 HIRES:PAPER 2: INK 0
 30 CURSET 120,100,3: REM centre cursor
 40 C=120: R=100: REM initial column and row
 50 FOR X=1 TO 360*10 STEP 10: REM 10 turns
 60 P=X*(PI/180): REM degree to radian
    conversion factor
 70 Q=P*(3/2): REM Q is "arm" increment
 80 A=C+Q*COS(P): REM column
 90 B=R+Q*SIN(P): REM row
100 CURSET A,B,1: REM set point
110 NEXT
```

Line 40 sets the cursor variables C and R to point to the centre of the screen. Line 50 determines how many turns there will be, 10 in this case – there are 360 degrees in a full circle. Line 60 converts degrees (X) to radians (P) for the functions SIN and COS. Line 70 introduces a factor which determines the spacing of the 'arms'. If you change line 70 to

```
70 Q=P*(P/10)
```

you can get some different types of spirals.

Lines 80 and 90 calculate each row and column position for the next pixel to be set. If you omit the STEP 10 in line 50, the detail is made much finer.

It is a relatively simple matter to convert this routine to draw circles. The advantage over the built-in CIRCLE command is that you can, to some extent, get round the problem of oval circles with a bit of tinkering. You can also use a STEP to make the periphery finer or coarser. Of course, the routine is a lot slower than the CIRCLE command.

```
10 REM simple circle routine
20 HIRES: PAPER 6:INK 0
30 C=120: R=100: V=PI/180: RA=20: REM RA is
   radius
40 FOR X=1 TO 360
50 P=X*V
60 A=C+RA*COS(P)
70 B=R+RA*SIN(P)
80 CURSET A,B,1
90 NEXT
```

Messages on the high-resolution screen

Putting characters on the HIRES screen is a bit more complex than using the drawing commands. The function you need for this is CHAR. This command requires three arguments, as in

```
100 CHAR A,S,FB
```

FB you have met before (see CURMOV, CURSET and CIRCLE). A is the ASCII code of the character you want and S is the character set to be used. If S is 0 then the normal character set is used – this includes any characters you have redefined. When S is set to 1, the alternate character set (teletext) is taken. Most of the time you will want to display strings of letters on the HIRES screen using CHAR. This requires a number of steps. If you wanted to put up the message 'PRESS SPACE TO CONTINUE', you would have to use a routine like this:

```
 10 HIRES
100 A$="PRESS SPACE TO CONTINUE"
120 S=0: FB=0: REM normal character set and
    letters in foreground colour
130 FOR K=1 TO LEN(A$)
140 L$=MID$(A$,K,1): REM a letter
150 L=ASC(A$): REM its ASCII code
```

```
160 CHAR L,S,FB: REM display it
170 CURMOV 10,0,0: REM move cursor
180 NEXT
```

Note especially line 170. Unlike PRINT in TEXT mode, CHAR does not automatically move the cursor once a letter has been displayed (sent to the screen). You have to control the movement of the cursor in your program. This gives you a wide range of possibilities. If you change line 170 to read 'CURMOV 0, 10, 0' you will find that the message runs down the screen. To make it run diagonally, you would use 'CURMOV 10, 10, 0'. CHAR therefore allows you to alter the spacing between letters, e.g. 'CURMOV 20, 0, 0' will put long spaces between each letter. Using CHAR, it is easy to format HIRES text in complex and appealing ways (words in circles, triangles, etc.).

POINT

POINT is an instruction that simply returns one of two numbers. It needs two arguments, these being C and R for Column and Row. It is used to find out the colour of a point or pixel on the HIRES screen. For example, if you wanted to know if the point 120,100 (the centre of the screen) was in the foreground colour, you would write something like this:

```
100 C=120: R=100
110 P=POINT(C,R)
120 IF P=0 THEN 140: REM background
130 IF P=-1 THEN PING: GOSUB 9000: REM
    foreground
```

Here, POINT puts a value on P. P will be 0 if the point at 120,100 is in the background colour, −1 if it is in the foreground colour (i.e. FALSE or TRUE). This function is of particular use for detecting collisions between moving objects on the screen. It is rather limited, however, and you would have to use a more complex sequence of calculations and PEEKs to find, for example, the exact value of the contents of a cell (byte) on the screen. POINT only looks at the pixels, i.e. operates at the bit level; PEEK gives you access to whole bytes.

FILL

FILL is the last of the HIRES commands. It is also the hardest to understand and use. FILL takes three arguments: NR, NC and AC (for example). These are Number of Rows, Number of Cells and ASCII code. What it does is to fill NR rows by NC cells with the number AC. You could do the same thing using nested FOR . . . NEXT loops and POKE, but FILL works very quickly. One of its best uses is in quickly altering the INK or PAPER colours on part of the HIRES screen.

Here's a routine that uses FILL to put vertical stripes of the colours available on the HIRES screen. This is handy for testing the colour balance of your TV and seeing how FILL works:

```
1Ø HIRES
2Ø CURSET Ø,Ø,Ø: REM cursor to top left
3Ø FOR COLOUR=Ø TO 7: REM 8 colours
4Ø CURSET COLOUR*3Ø,Ø,Ø
5Ø FILL 2ØØ,1,COLOUR+16
6Ø NEXT COLOUR
```

Line 40 sets the column positions at 0, 30, 60 90, 120, etc. for all the colours. Line 50 FILLS 200 rows (i.e. the whole screen, from top to bottom) of one cell (column) with the colour code given by adding the variable COLOUR to 16. The reason for adding 16 is that we are using the same values here as in the TEXT section, where 16 is black PAPER, 17 is red PAPER, 18 is green and so on.

The same effect can be achieved using POKE to get the PAPER attributes on to the HIRES screen:

```
1Ø SS=4Ø96Ø: REM HIRES screen origin
2Ø NC=7: AN=16: REM no. colours -1, colour
   code offset
3Ø HIRES
4Ø FOR RO=SS TO SS+2ØØ*4Ø STEP 4Ø
5Ø REM from start to end of screen in steps of
   4Ø cells (i.e. row by row)
6Ø FOR K=Ø TO NC
7Ø POKE RO+5*K,K+AN
8Ø NEXT K
9Ø NEXT RO
```

A table of what this does will help you to understand the HIRES screen:

ADDRESS	40960								40965							
BIT NUMBERS	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
(ROW 1) (BITS)	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	1
BINARY VALUES	128	64	32	16	8	4	2	1	128	64	32	16	8	4	2	1
DECIMAL CONTENT	16								17							
COLOUR	BLACK PAPER (continues . . . until)								RED PAPER							

This table shows how each bit of two screen addresses is set by the example program given above. Location 40960 (the screen origin) is POKEd with 16, setting bit 4. This is then taken as the PAPER attribute BLACK. This background continues to affect the addresses in that first row until address 40965. This is POKEd with 17, setting bits 4 and 0, giving red PAPER. The process continues across the top line; address 40970 will have bits 4 and 1 set, giving a background colour of green and so on. This sequence of events is repeated for all 200 rows, giving the effect of eight vertical stripes down the screen, of all the colours available.

If you wanted to POKE the screen with PAPER attributes you would have to alter line 70 to read:

```
70 POKE RO+5*K,K
```

This would put the values 0 (black INK), 1 (red INK), 2 (green INK) etc. into the same locations as before. That is, only bits 0, 1 and 2 are affected by these values, so the attribute is taken as INK, not PAPER. Note that INK values can be converted to PAPER values by adding 16; here is a table for quick reference:

POKE		
Ink	*Paper*	*Colour*
0	16	Black
1	17	Red
2	18	Green
3	19	Yellow
4	20	Blue
5	21	Magenta
6	22	Cyan (light blue)
7	23	White (buff)

Don't forget that you can't use the numbers in the second column directly with the PAPER command, they're the actual values that you would POKE on to the screen to get the relevant effect. Note too that you can add 128 to all of these to get the same effect.

Writing in circles

As mentioned earlier, it is fairly easy to devise routines to display text in unusual ways on the HIRES screen. The first example shows a routine which will display a message in a circle:

```
 10 A$="HI THERE EVERYBODY"
 20 A$=A$+" "
 30 L=LEN(A$)
 40 IF L/2 <> INT(L/2) THEN L=L+1: A$=A$+
    CHR$(32)
 50 ST=360/L
 60 C=120: R=100: V=PI/180: FB=1
 70 RA=L/2*PI
 80 HIRES
 90 D=0
100 CURSET C,R,0
110 FOR AL=1 TO L
120 X=D*V
130 A=C+RA*COS(X)
140 B=R+RA*SIN(X)
150 CURSET A,B,FB
160 LC=ASC(MID$(A$,AL,1))
170 CHAR LC,0,1
180 D=D+ST
190 NEXT
```

As the formulae for spirals are so similar to those for circles, it is easy to have a message displayed in a spiral by changing lines 130 and 140 and adding a variable to increase the effective radius at each point. One way of doing this would be to add:

```
125 Q=X*(9)
13Ø A=C+Q*COS(X)
14Ø B=R+Q*SIN(X)
```

Contours

'Contour mapping' can produce some interesting and aesthetically pleasing pictures. This program shows how to use numeric arrays and how to use the DRAW function of the Oric in HIRES mode. The program draws a polygon inside another, then proceeds to fill the space between with 'contour' lines. Fig. 10.2 shows the general principle. You can change the numbers in the DATA statements at the end of the program to make your own shapes but you must bear in mind the following points:

1. N1 and N2 give the number of points for the outer and inner shapes respectively.
2. The points must be given in the order X, Y (column, row) and must be given in clockwise order in the DATA statements.
3. The inner shape must have fewer points than the outer.

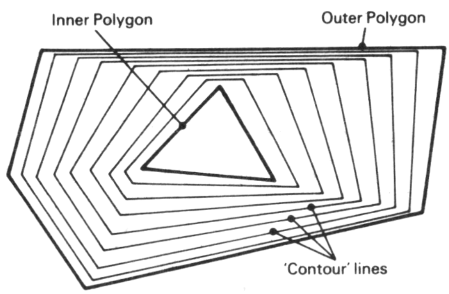


Fig 10.2 Contours

You can also alter the value of S in line 420. This gives the detail of the contours; 30 will produce lines close together, a value of 5 makes the contour line spacing much wider.

You might find some of the other routines useful; for example, the routine in lines 330 to 370 could be modified to rank data in much larger arrays prior to some statistical calculations.

A detailed explanation of the program would occupy several pages and is beyond the scope of this book. However, there are one or two points that bear discussion. The most important of these is that although the DATA are given as the absolute screen coordinates of each point, the points are linked by DRAW, which is a relative command. That is, you cannot DRAW from one point to

another; you have to 'tell' DRAW to make a line which is defined by moving a certain number of columns and a certain number of rows. If you have two points defined by the coordinates (X1, Y1) and (X2, Y2), and have just 'CURSET X1, Y1, 0', then to draw a line to (X2, Y2), you have to calculate the relative movement needed. This is done by 'DRAW X2 − X1, Y2 − Y1, 1'.

If you don't know the current coordinates of the cursor, these can be obtained from locations #219 and #21A for the X and Y coordinates respectively (see lines 120 and 240).

```
  5 REM contour drawing program
 1Ø HIMEM #17FF: REM #97FF for 48K users
 2Ø HIRES: PAPER 2: INK Ø
 25 REM N1=number of 'outer' points
 26 REM N2=number of 'inner' points
 3Ø N1=8: N2=3
 35 REM X() holds column coordinates
 36 REM Y() holds row coordinates
 37 REM D() holds differences
 4Ø DIM X(2,N1),Y(2,N1),D(N1,N2)
 45 REM read & draw outer shape
 46 PRINT "READING AND DRAWING OUTER SHAPE"
 5Ø FOR K=1 TO N1: READ X(1,K),Y(1,K): NEXT
 6Ø CURSET X(1,1),Y(1,1),Ø
 7Ø FOR K=1 TO N1-1
 8Ø X1=X(1,K): X2=X(1,K+1): Y1=Y(1,K):
    Y2=Y(1,K+1)
 9Ø A=X2-X1: B=Y2-Y1
1ØØ DRAW A,B,1
11Ø NEXT
12Ø A=PEEK(#219): B=PEEK(#21A)
13Ø DRAW X(1,1)-A,Y(1,1)-B,1
135 REM read & draw inner shape
136 PRINT: PRINT "READING AND DRAWING INNER
    SHAPE"
14Ø FOR K=1 TO N2
15Ø READ X(2,K),Y(2,K)
16Ø NEXT
17Ø CURSET X(2,1),Y(2,1),Ø
18Ø FOR K=1 TO N2-1
19Ø X1=X(2,K): X2=X(2,K+1)
2ØØ Y1=Y(2,K): Y2=Y(2,K+1)
21Ø A=X2-X1: B=Y2-Y1
22Ø DRAW A,B,1
23Ø NEXT
24Ø A=PEEK(#219): B=PEEK(#21A)
25Ø DRAW X(2,1)-A,Y(2,1)-B,1
29Ø REM calculate differences
295 PRINT: PRINT "CALCULATING DIFFERENCES"
3ØØ FOR D=1 TO N1: FOR D2=1 TO N2
```

```
310 D(D,D2)=ABS(X(1,D)-X(2,D2))+ABS(Y(1,D)-
    Y(2,D2))
320 NEXT: NEXT
325 REM rank differences
326 PRINT: PRINT "RANKING DIFFERENCES"
330 FOR R=1 TO N1: FOR B=1 TO N2
340 BN=0: FOR C=1 TO N2
350 IF D(R,C)>BN THEN BN=D(R,C): CN=C
360 NEXT: D(R,CN)=N2-B+1
370 NEXT: NEXT
390 REM draw contours
395 PRINT: PRINT "DRAWING CONTOURS"
400 A=X(1,1)): B=Y(1,1): OX=A: OY=B
410 CURSET A,B,1
420 S=30: REM step size: low=coarse
430 FOR I=1 TO S-1
440 FOR F=0 TO I-1: ST=ST+1/S: NEXT
450 FOR EP=1 TO N1
460 FOR SS=1 TO N2
470 IF D(EP,SS)=1 THEN J=SS
480 NEXT
490 A=X(1,EP)+(ST*(X(2,J)-X(1,EP)))
500 A=INT(A+.5)
510 B=Y(1,EP)+(ST*(Y(2,J)-Y(1,EP)))
520 B=INT(B+.5)
530 DRAW A-OX,B-OY,1
540 OX=A:OY=B: NEXT: ST=0
550 NEXT
560 PRINT: PRINT "ALL DONE":
570 END
590 REM data block in form X,Y
595 REM outer then inner, clockwise
596 REM outer
600 DATA 10,170,60,90,10,20,100,40,210,
    10,230,130,200,190,130,150
605 REM inner
610 DATA 110,110,130,80,150,110
```

Finally, Fig. 10.3 is provided as a reference guide to the HIRES screen.

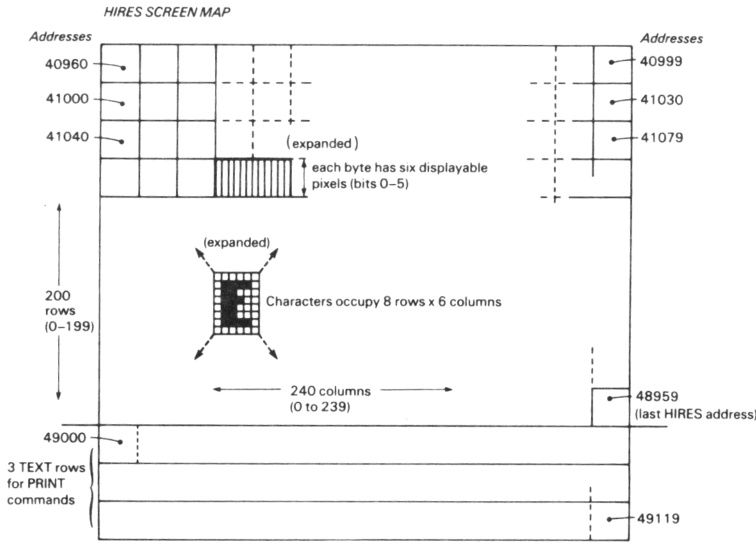


Fig. 10.3 HIRES screen map

16k owners may subtract 32768 (8000 hex) from these addresses – see Appendix F.

Quite what happens between 48959 and 49000 in HIRES is anybody's guess. (These are the last address of the graphics section of the HIRES screen and the first byte of the TEXT part of the HIRES screen respectively.)

11. Serial Attributes – Colours

The Oric uses a system known as 'serial attributes' to define the colours of the foreground and background of the TEXT and HIRES screens. This is not the easiest system to understand, but is fairly easy to use once mastered, so perseverance will pay off.

It is important to realize and remember that in either TEXT or HIRES mode, one cell (byte) may contain either an attribute or a pattern, but not both. That is, the contents of a screen address will either cause something to be displayed on the screen or will affect the INK or PAPER colours of items to its right (on the same row). Once PAPER or INK values have been set, they will continue to affect all items shown on that line until changed by another attribute in another address on the same row. There are 200 such rows on the HIRES screen and 27 in TEXT and both LORES modes.

Normally, locations #026B and #026C hold the PAPER and INK colours which are put in the first two columns of each row to set the global INK and PAPER colours on the screen. These two left-hand columns are 'protected', that is, you should not be able to use them. However, as you have seen, it is possible to PLOT in the second of them, i.e. affect the INK colour, and this can be regarded as another problem with the Oric's interpreter. If you 'send' CHR$(29), which is the same as CTRL], you can 'unprotect' these columns. You can only do this in TEXT mode to get an effect. After sending this non-printing character, all PRINT commands will start items in the first screen column. Since there is neither an INK nor a PAPER attribute to the left of such items, they will appear in monochrome – white INK on black PAPER.

Attributes

An attribute is simply a value of less than 32 or between 128 and 160 in a screen cell or byte. When the Oric finds such a value while sending things to the screen, it doesn't display a pattern (such as a letter, number or group of pixels), but alters the INK or PAPER colours to the right of that cell, as far as the end of that screen line or as far as the next attribute which alters the same aspect of the display. You should recognize the important values from the discussion of control codes in Chapter 6.

Attributes may be PRINTed or PLOTted or POKEd on to the TEXT and (with less effect) the LORES screens. Using the HIRES screen, attributes have to be placed using either POKE or FILL. For example, to split the HIRES screen into two different PAPER colours, the following compares POKE and FILL.

```
 10 HIRES: INK 0: PAPER 2
 20 REM using FILL
 30 CURSET 120,0,0
 40 FILL 200,1,17
 50 GET A$
 60 REM POKE
 70 HIRES: INK 0: PAPER 2
 80 FOR I=40978 TO 48999 STEP 40
 90 POKE I,17
100 NEXT
110 GET A$: TEXT: LIST
```

You will see from this how much faster FILL is and how it can be used to affect large areas of the screen quickly. POKE you would use to affect small, precisely defined areas.

The following tables show why the number 32 is so important: numbers below this figure have bits 5 and 6 set to zero, and it is these bits which determine whether a screen byte is taken as a pattern or an attribute. Bit 7 indicates 'inverse'.

BITS	7	6	5	4	3	2	1	0	
DECIMAL VALUES	128	64	32	16	8	4	2	1	
SOME BINARY PATTERNS AND THEIR EFFECTS	0	0	0	1	1	1	1	1	ASCII code 31
	0	0	1	0	0	0	0	0	ASCII code 32 – a space
	0	1	0	0	0	0	1	0	ASCII code 66 – the letter 'B'
	0	0	0	0	0	0	0	1	ASCII code 1 – gives Green INK
	0	0	0	1	0	0	1	0	ASCII code 18 – gives Yellow PAPER
	128	64	32	16	8	4	2	1	

PAPER and INK attributes are, of course, independent. This means that you can change either of them on the same line without affecting the other. You could easily change from red to blue PAPER and from green INK to black several times on the same line for instance. One point that you do have to bear in mind, however, is that *any* attribute loses you a display cell. To reset the INK and PAPER attributes for any given point, you will use two bytes of screen memory. In TEXT mode, this means that you will have two blank spaces to the left of an item whose attributes you want to alter from its surrounds. This is of particular importance if you're PRINTing or PLOTting a string whose first two characters are non-printing, attribute-controlling characters. You would have to be quite careful, when moving such a block of 'text' around the screen, that the two leftmost invisible attributes which define INK and PAPER for the string don't overwrite other attributes or patterns on the screen.

An example

Here's an example of POKEing INK and PAPER attributes on to the screen between words to get different INK and PAPER colours on the same line. Note that each word has two spaces after it, for the INK and PAPER attributes.

```
10 CLS: INK 0: PAPER2
20 FOR J=1 TO 5: PRINT "WORD  ";: NEXT: REM 2
   spaces after the D in "WORD"
30 REPEAT
40 FOR OT=0 TO 39 STEP 6
50 POKE 48040+OT, INT(RND(1)*7)
60 POKE 48041+OT, INT (RND(1)*7+16)
70 NEXT
80 WAIT 100
90 UNTIL KEY$ <>"": REM press a key to stop
```

To do this sort of thing you need to work out the length of each word, its address in RAM and then POKE the INK or PAPER attributes to the left of the word. It doesn't matter which of these attributes comes first, what's most important to remember is that one space on the screen can only ever contain a character *or* an attribute and that the effects of an attribute carry on along that line until reset by another attribute.

Fig. 11.1 shows how the letters and attributes appear in the Oric's TEXT memory.

RUN the program below. The table shows each address of the first twelve cells of the top screen row. The second row of the table gives the contents of each address, the third row shows how the Oric uses the contents of a screen address either to display a character or to set the attributes of items to the right of a TEXT address or cell whose contents are less than 32.

```
10 PAPER 1: INK 0: CLS
20 PRINT "WORD  WORD": REM 2 spaces separate the
   words
30 POKE 48046,3: REM yellow paper
40 POKE 48047,22: REM cyan ink
50 FOR A=0 TO 11
60 PRINT PEEK(48040+A);: REM display contents
   of 1st 12 cells of line 1
70 NEXT
```

ADDRESS	48040	48041	48042	48043	48044	48045	48046	48047	48048	48049	48050	48051
CONTENTS	1	16	87	79	83	68	3	22	87	79	82	68
MEANING	RED PAPER	BLACK INK	W	O	R	D	YELLOW PAPER	CYAN INK	W	O	R	D

Fig. 11.1 How letters and attributes appear in the TEXT screen memory

Coloured letters in HIRES

FILL can be useful for making text messages on the HIRES screen appear in appealing ways. The next example displays the letter E on the HIRES screen, then uses FILL to alter the INK attributes for each of the eight rows which go to make up the character display. This technique could be extended for longer messages.

```
 1Ø HIMEM #97FF: REM #17FF for 16K machines
 15 DEF FNR(X)=INT(RND(1)*X)
 2Ø HIRES: CURSET 2Ø,Ø,Ø
 3Ø CHAR 69,Ø,2
 35 X=7
 4Ø REPEAT
 5Ø FOR K=Ø TO 7
 6Ø CURSET 1Ø,K,Ø
 7Ø FILL 1,1,FNR(X)
 8Ø NEXT
 9Ø UNTIL KEY$<> ""
1ØØ TEXT: LIST
```

This program also shows how to use the Oric's facility for user-defined functions. Line 15 sets up a function that will generate random integers in the range 0 to X. All you have to do is to pass the variable X (the highest number that you want) to it and then use it as shown. This saves having to write numerous lines like 'C=RND(1)*10+1', for example. Another important point is that you should always use CURSET after a FILL to position the cursor precisely where you want it. The Oric can put the cursor in unexpected places after FILL!

Bytes, bits and pixels

On the HIRES screen, attributes again occupy one cell, but you have to bear in mind that each cell consists of eight bits. That is, each byte of screen memory contains eight binary positions. On the Oric, only six bits of any screen byte can be set (i.e. appear visible). This means that with 40 character columns across the screen, you have 240 pixels (6×40). A pixel is a picture element. Only the last six bits of any screen byte can be displayed; bit 7 is for 'inverse' and bit 6 (together with bit 5) helps determine whether a byte is an attribute or a pattern. If bits 5 and 6 are both 0, then the number is less than 32, so the byte must be an attribute. If either bit is set, the number is greater than 32 and is displayed as a pattern. This explains why the Oric is limited to 128 characters in its ASCII set (other machines can have 255 such characters, which allows users to define character shapes without having to alter any of the standard character set). It also explains why the character set is repeated from 128 to 255 – the only difference between the binary codes for the numbers 128 to 255 and 0 to 127 is that the high bit is set. As this is used for inverse display, the Oric 'thinks' that (as far as 'patterns' are concerned) 254 is the same as 254 − 128 = 126.

PRINT and PLOT

However, none of this really explains why PRINT and PLOT have different effects where the ASCII set is concerned. With ASCII codes 0 to 7, PRINT CHR$ gives the control codes such as keyclick on/off, while PLOT CHR$ gives the INK attributes. In this instance, PLOT must be setting the high bit before display. With ASCII codes in the range 128 to 255, PRINT CHR$ has the same effect as PLOT with lower numbers, but PLOT CHR$ actually makes the attribute cell visible on the screen as a pinkish square – very strange.

As above, the Oric does not handle attributes consistently. PRINT and PLOT can have different effects. If you 'set' the high bit (bit 7) of a screen byte's contents, the local INK and PAPER colours should be reversed. This is a very useful way to highlight messages or display a pattern without having to use preceding attributes which waste a screen cell. However, PRINT resets bit 7 to 0, as does CHAR. PLOT is the only display command with which you can use this trick. To demonstrate this, try:

```
10 FOR I=0 TO 7
20 CLS: INK I: PAPER 2
30 PRINT "A"
40 PLOT 2,0,65+128
50 GET A$: REM press a key for next colour
   complement
60 NEXT
70 INK 0
```

This will show you what inverting the display does – it gives you complementary colours, allowing you limited colour changes without having to use attributes. The number 65 in line 40 is the ASCII code for the letter A and adding 128 sets the high bit (bit 7) as shown in Fig. 11.2.

	Normal A								Inverse A							
BITS	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
BITS SET	0	1	0	0	0	0	0	1	1	1	0	0	0	0	0	1
DECIMAL VALUES	128	64	32	16	8	4	2	1	128	64	32	16	8	4	2	1

Fig. 11.2 To show how adding 128 to the ASCII code of the letter A sets bit 7

CHAR

One thing to watch out for when using CHAR is that you don't try to display a character on the HIRES screen where it would overwrite an attribute. If this does happen, only part (if any) of the character will appear. The following program shows this by FILLing vertical rows with INK and PAPER attributes, then putting random characters on the screen in randomly selected positions. You will see how any character with part of its matrix overlapping an attribute is distorted.

```
 1Ø DEF FNR(X)=INT(RND(1)*X)+1: REM random
    number function
 2Ø HIRES: PAPER 7: INK 1
 3Ø CURSET Ø,Ø,Ø
 4Ø FOR K=Ø TO 2ØØ STEP 1Ø
 5Ø CURSET K,Ø,Ø
 6Ø FILL 2ØØ,1,FNR(7): REM fill 2ØØ rows by 1
    column with ink attribute
 7Ø CURSET I+8,Ø,Ø: REM move cursor 1 byte
    (8 bits) to right
 8Ø FILL 2ØØ,1,FNR(7)+15: REM fill 2ØØ rows by
    1 column with random paper
 9Ø NEXT K
1ØØ REPEAT
11Ø CURSET FNR(231),FNR(181),Ø: REM select
    random position
12Ø CHAR FNR(26)+64,Ø,1: REM display random
    character
13Ø UNTIL KEY$ <>"": REM press any key to stop
14Ø TEXT: LIST: REM select text and list program
    for editing
```

Escape sequences

'Escape sequences' are the 'official' way to change the INK or PAPER of PRINTed strings. To use them you have to 'send' the ASCII code for ESCape which is CHR$(27), followed by the relevant control code. This is a character from A to ← (left arrow) (as shown in the incomprehensible table of Appendix B of the Oric manual).

For example:

```
1Ø PRINT CHR$(27);"LTHIS SHOULD BE FLASHING"
```

What *should* happen here is that once the ESCape code (CHR$(27)) has been 'sent', the Oric looks for a control code. In this example the letter L is put as the first character of the message to be printed and will not actually appear, but be used as a code to affect the next set of characters to be PRINTed.

In fact, because of a slight problem in the way the Oric handles ESCape sequences, this example doesn't work. If you use escape sequences, you must

make sure that CHR$(27) is not the first character on a line. To make the above example work properly, you would have to add line 5:

```
5 PRINT " ";
```

or

```
5 PRINT CHR$(32);
```

That is, put a space as the first character of the line (32 is the ASCII code for a space).

To use the table in the manual, simply send CHR$(27), then the letter for the characteristic you want. You can do this in two ways, either by putting the relevant letter as the first character of the string to be PRINTed, or as in the following:

```
 5 PRINT CHR$(32);
1Ø PRINT CHR$(27);"L";"THIS WILL FLASH."
```

It doesn't matter whether you use the semicolon or the string concatenation symbol (+) to link the escape codes and the strings to be printed. Appendix D gives the ASCII codes as well as the ESCape codes for all the attribute functions.

12. Making Noises and Music

The Oric has four predefined sounds that you can use in your programs. These are EXPLODE, PING, SHOOT and ZAP. To make the sounds, enter the word at the keyboard or use it as a statement in your program like this:

```
100 SHOOT
```

These sounds can be useful because you don't have to work out in detail what sort of sound you want or how to get it. You can use them in conditional statements like:

```
290 IF NL=1 THEN ZAP ELSE SHOOT
```

You cannot change any of the sounds made with these commands.

SHOOT is a bit like EXPLODE. PING is the same as CTRL G or PRINT CHR$(7). These can all be used without affecting the speed of a program. ZAP is a little different because, while it is sounding, the program will pause. This is due to the fact that the sound of ZAP is altered continually during its production.

Fortunately, you're not limited to these sounds and can even get your Oric to play fairly complex tunes and make a wide variety of sounds and noises. In fact, you can make it produce up to six different noises at a time. One nice point is that making a sound doesn't interfere with anything else. Some microcomputers cannot do anything else while a noise is being produced, which can slow down programs considerably.

The Oric has three 'channels' through which you can play pure tones (sine waves). There is also a noise channel which can be added to any or all of the sound channels; this allows you to modify notes to be played.

There are three basic words which deal with sounds. These are SOUND, MUSIC and PLAY. PLAY is usually used when the volume settings of the other commands are set to zero. SOUND and MUSIC are very similar in their effects.

Sound

SOUND takes three arguments: channel, period and volume.

Channel

The channel can be a number in the range 1 to 6 and tells the Oric which channel should be used for the sound. This means that you can have up to three sounds produced at the same time by directing the sounds specified through the three channels at the same time. The numbers 1, 2 and 3 refer to

the tone channels. Values 4 to 6 specify which tone channel (1 to 3) noise should be added to: value 4 adds noise to tone channel 1; value 5 adds it to 2; value 6 to 3.

Pitch

Period determines the pitch (frequency) of the sound. It can be any number from 1 to 65,535. 1 is the highest tone. The Oric does something funny with the period values. If you try the next program, you will hear the tones falling, until the period is about 4000, then the pitch will suddenly rise and start falling again. That is, the tones from 1 to 4096 are repeated from 4096 to 8192 and so on in rough multiples of 4096 up to 65,535.

Turning the keyclick off

You will find that with some of the programs here and, indeed, with your own, it is important to turn off the keyclick when using the sound facilities of the Oric. This is because the keyclick is generated by the same sound chip that is used to make the sounds: the keyboard scanning routine will detect a key-press, then direct the chip to produce a click, even in the middle of your sound. This problem can be avoided by using the keyclick toggle CTRL F or PRINT CHR$(6). However, since CTRL F is under user control at most points during your program, a better method is needed.

Many of the 'toggle' factors are held in address #26A (618 decimal). That is, by poking location 618 with various values you can control things like printer on/off, cursor on/off, double-height character printing on/off and so on. To turn off the keyclick – no matter whether it's already on or off – you have to set bit 3 to 0. This can be done by ORing the contents of #26A with 8 (00001000 in binary) i.e. 'POKE 618, PEEK(618) OR 8'. This will *always* turn the keyclick off and is therefore far more useful than simply toggling the click switch. Turning the click back on needs the opposite: 'POKE 618, PEEK(618) AND NOT 8'. Similar techniques can be used to ensure the status of any of the functions which are defined in that particular address.

Volume

Volume can be set in the range 0 to 15. 15 is rather loud. If you set the volume to 0, no sound will be produced until a PLAY command is encountered.

```
10 REM demonstration of sound
20 CLS
30 FOR P=1 TO 65535 STEP 100
40 SOUND 1,P,7: REM channel 1, period, volume 7
50 PRINT P;
60 WAIT 10
70 NEXT
80 PLAY 0,0,0,0
```

There are a few points to note here. Line 60 determines the length of the sound produced – try changing it from 10 to 1 or 100. When using SOUND,

you must remember to turn off the last sound made with a PLAY 0,0,0,0 statement (line 80). This shuts off all the tone and noise channels.

The next program shows how to make more than one sound at a time. This can be used to play chords in music, for example, or to make more complex tones.

```
10 REM multiple channels
20 PLAY 7,0,0,0: REM open all 3 tone channels
30 FOR P=1 TO 4000
40 SOUND 1,P,7: REM channel 1, period, volume 7
50 SOUND 2,P+20,7: REM channel 2, period+20,
   volume 7
60 SOUND 3,P+40,7: REM channel 3, period+40,
   volume 7
70 WAIT 10
80 NEXT
90 PLAY 0,0,0,0: REM all sound off
```

You will notice that using more than one channel requires an extra command; line 20 opens channels 1, 2 and 3 (see the table in the PLAY section). In this example, the tones played through the three channels are all different, which is done by increasing the period values for each channel in lines 50 and 60. This is one technique that could be used to produce harmonies or chords.

MUSIC

MUSIC is very similar to SOUND, but it allows you to define the periods of tones in relation to musical notation.

It is quite easy to translate sheet music to numbers; an example is given later. MUSIC takes four arguments; channel, octave, note, volume.

Channel

Channel is used as for tone, but should only be between 1 and 3, since only tones are being made.

Octave

There are 7 octaves (0 to 6); 0 is the lowest.

Note

Note values can be between 1 and 12. 1 is the note C; 2 is C sharp; 3 is D; 4, D sharp and so on: each number is one semitone higher than the previous number.

Volume

If the volume is set to 0, then, as with SOUND, no sound will be played until a PLAY command which 'enables' the relevant channels is met in your program.

The reason MUSIC is so similar to SOUND is that once you have given MUSIC all the information it needs, it looks up the correct SOUND to produce in a table then jumps to the sound-producing routine to make a sound of the proper musical octave, note, etc. In fact, you could work out such a table yourself and never use MUSIC at all. Of course, MUSIC is a lot easier to use since it takes a lot of the hard work out of transcribing sheet music into a form that the Oric can 'understand'.

Using MUSIC

The easiest way to use MUSIC is to put the necessary values for the notes, their octaves, etc., into DATA statements or arrays. Here's an example which plays the scale of C through all the octaves.

```
 1Ø REM scale of C
 2Ø DATA 1,3,5,6,8,1Ø,12
 3Ø REM C,D,E,F,G,A,B
 4Ø REPEAT
 5Ø READ N: REM read note value
 6Ø MUSIC 1,O,N,7: REM channel 1, octave, note,
    volume 7
 7Ø WAIT 2Ø: REM note length
 8Ø IF N=12 THEN RESTORE: O=O+1
 9Ø REM if last note, restore data pointer to
    first note, add 1 to octave
1ØØ UNTIL O>6: REM all octaves done
11Ø PLAY Ø,Ø,Ø,Ø: REM turn off sounds
12Ø END
```

This program also shows how to repeat READing a set of DATA by using the command RESTORE. Notice that O (octave) is not defined at the start of the program and is therefore assumed by the Oric to be 0 when first met.

This example only used one value per note, but when transcribing sheet music to numbers you might want to specify the octave, note, its duration, volume and even channel. This would mean that each note would have to be given up to five numbers in a DATA statement. Each DATA statement might look like this: 'DATA 1,2,7,3,100: REM channel 1, 3rd Octave, note F♯, vol 3, length 100'. This would involve a lot of work with pencil and paper but can be very rewarding.

Sheet music

Figs 12.1, 12.2 and 12.3 show the notes and their positions on the musical staves together with the name of each note and the number you would use to tell the Oric which note to play.

The next program shows the notes in the key of C for the treble clef, and is given here as an example of several of the points made in previous chapters.

```
1Ø HIMEM #17FF: HIRES: PAPER 2: INK Ø
2Ø REM #97FF for 48K
```

```
  3Ø GOSUB 1ØØØ: REM draw staves
  4Ø GOSUB 2ØØØ: REM define note shape
  5Ø GOSUB 3ØØØ: REM draw notes, numbers
  6Ø :           REM and play note
  7Ø REM all done
  8Ø END
1ØØØ REM draw lines
1Ø1Ø CURSET 16,5Ø,Ø
1Ø2Ø NL=Ø:R=5Ø
1Ø3Ø REPEAT
1Ø4Ø CURSET 16,R,Ø
1Ø5Ø DRAW 22Ø,Ø,1
1Ø6Ø NL=NL+1;R=R+11
1Ø7Ø UNTIL NL=5
1Ø8Ø RETURN
2ØØØ REM define note shape
2ØØ5 OS=6144: REM 16K (38912 for 48K)
2Ø1Ø DATA 4,4,4,4,28,28,28,28
2Ø2Ø S=OS+(96*8):E=S+7
2Ø3Ø FOR AD=S TO E
2Ø4Ø READ V:POKE AD,V
2Ø5Ø NEXT
2Ø6Ø REM put values into NV$()
2Ø7Ø REM and letters into NN$()
2Ø8Ø DATA 6,5,3,1,12,1Ø,8
2Ø9Ø DATA F,E,D,C,B,A,G
21ØØ FOR L=1 TO 7
211Ø READ NV$(L)
212Ø NEXT
213Ø FOR L=1 TO 7
214Ø READ NN$(L)
215Ø NEXT
216Ø RETURN
3ØØØ REM draw notes, letters, number
3Ø1Ø REM sound
3Ø2Ø NC=-9:NR=37: REM note column, row
3Ø3Ø N=1: O=4: REM start at F in 4th oct
3Ø4Ø REPEAT
3Ø5Ø N$=NN$(N)'GET NOTE LETTER
3Ø6Ø NC=NC+25:IF N/2<>INT(N/2) THEN NR=NR+7
     ELSE NR=NR+5: REM calculate new row
3Ø65 FX=Ø:FX=FX-(N$="E")-(N$="C")-(N$="F")+
     (N$="D")+2*(N$="G")
3Ø7Ø CURSET NC,NR+FX,Ø
3Ø8Ø CHAR 96,Ø,1: REM draw note
3Ø9Ø CURMOV 1Ø,1,Ø
31ØØ CHAR ASC(N$),Ø,1: REM draw letter
311Ø CURMOV 1Ø,Ø,Ø
312Ø NV$=NV$(N):L=LEN(NV$):NV=ASC(NV$)
```

```
3130 IF L=1 THEN CHAR NV,0,1: GOTO 3170
3140 CHAR ASC(LEFT$(NV$,1)),0,1: CURMOV
     6,0,0: CHAR ASC(RIGHT$(NV$,1)),0,1
3150 REM draw note numbers (>9)
3160 NC=NC+8: REM move to right+8
3170 TV=VAL(NV$): REM tone value
3180 MUSIC 1,O,TV,7: WAIT 100
3190 N=N+1: REM next note
3200 IF N>4 THEN O=3: REM drop an octave
3210 IF N=8 THEN EF=1: N=1
3220 UNTIL N=2 AND EF=1: REM second time around
3230 PLAY 0,0,0,0
3240 RETURN
```

NOTE	C	C♯	D	D♯	E	F	F♯	G	G♯	A	A♯	B
VALUE	1	2	3	4	5	6	7	8	9	10	11	12

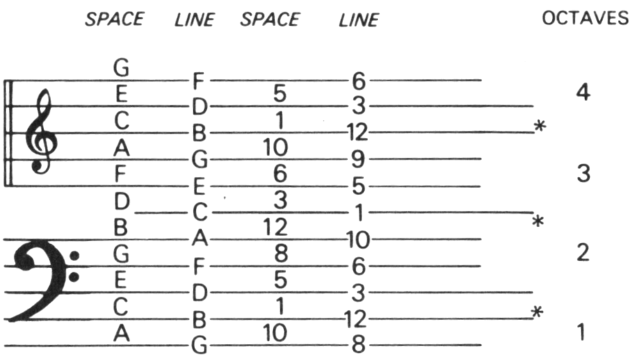


Fig. 12.1 Notes and their Oric values

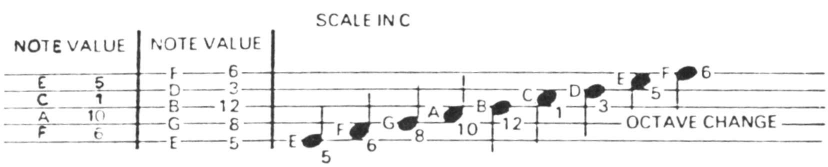


Fig. 12.2 The scale of C

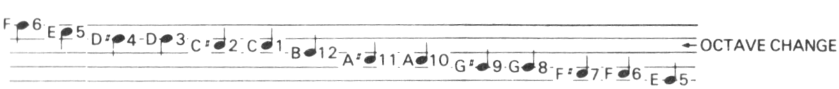


Fig. 12.3 Notes, stave positions and values

Note that in the subroutine at line 2000, in which ASCII character code 96 (the copyright symbol ©) is redefined as the note shape, the character set origin in memory is given as 6144. This figure is for a 16K system and you should add #8000 to it for 48K machines. The apparent discrepancy between this figure and that given in the TEXT section is simply that when HIRES is entered, the Oric moves the entire character set (both standard and teletext) from 46080 to 6144.

The variable FX is used as an adjustment to the row value of each note in order to place it exactly where it should be. A better program would be based around using CIRCLE for the body of the note and DRAW for the tail, but would require a longer and more complex algorithm and hence code.

Finally, here's a program that plays random notes, then plays a chord which consists of the notes one tone either side of the tone.

```
 1Ø REM chords
 15 R=RND(-PEEK(63Ø))
 2Ø DEF FNR(X)=INT(RND(1)*X)+1
 3Ø PLAY 7,Ø,Ø,Ø
 4Ø FOR N=1 TO 1Ø
 5Ø AN=FNR(8)+2
 6Ø O=FNR(6)
 7Ø MUSIC 1,O,AN,9
 8Ø WAIT 5Ø
 9Ø MUSIC 1,O,AN-2,7
1ØØ MUSIC 2,O,AN,7
11Ø MUSIC 3,O,AN+2,7
12Ø WAIT 5Ø
13Ø NEXT
14Ø PLAY Ø,Ø,Ø,Ø
```

PLAY

PLAY is the hardest command in this section to get to grips with, yet it offers great control over the Oric's sound-producing capabilities. As mentioned above, PLAY is used when the volume setting for either SOUND or MUSIC has been set to 0. Under some circumstances, PLAY may be needed to open channels for sound after they have been closed in an earlier part of a program.

PLAY enables channels for tones and other sounds to be produced, allows you to mix noises with tones, vary the quality or timbre of any sound produced and so on. PLAY takes four arguments: tone channels, noise channels, envelope and envelope period.

Tone and noise channels

The first two arguments are very similar; they 'enable' channels according to the following table:

Value	*Channels opened*
0	none
1	1
2	2
3	1 and 2
4	3
5	1 and 3
6	2 and 3
7	all

To use the first two arguments of PLAY, you first have to decide which channel numbers you want sound to be directed through. You then use a MUSIC or SOUND command to get the frequency or pitch you want and set the volume in the SOUND or MUSIC instructions to 0. Finally, you use PLAY to enable the channels specified in the relevant SOUND or MUSIC commands. This lets you define tones at one point in your program which are actually produced in another. It also allows you to control which channels should have noise added to them by using the second argument to enable noise channels in conjunction with the sound channels. The best way to find out about these first two values is to experiment.

Using the tone and noise enabling system means that you can have up to six sounds being produced at more or less the same time. Here's an example of the Oric going flat out to make as much noise as possible.

```
  1 REM noisy Oric
 1Ø N=1
 2Ø PLAY 7,Ø,Ø,Ø: REM open all 3 tone channels
 3Ø FOR T=1 TO 38ØØØ STEP 1Ø: REM from high to
    low pitch/frequency
 4Ø FOR CH=1 TO 3: REM for all 3 channels
 5Ø SOUND CH,T+1ØØ,Ø: REM load with sound
 6Ø MUSIC CH,O,N+3,Ø: REM load with music
 7Ø N=N+1: IF N>9 THEN N=1: O=O+1: REM don't let
    N exceed 12 (9+3): reset to Ø, increase
    octave
 8Ø IF O>6 THEN O=Ø: REM 6 is highest octave
 9Ø NEXT: REM next channel
1ØØ PLAY 7,Ø,1,5ØØØØ: REM all loaded - do sounds
11Ø NEXT: REM next tone
12Ø PLAY Ø,Ø,Ø,Ø: REM all done, turn it off
```

Envelope

The third argument for PLAY is envelope. This allows you to define the 'shape' of the tones produced. There are seven possible values for the envelope factor (1 to 7). The first two of these give sounds which are 'finite'. This means that a tone will start, last a certain time, then finish. Putting numbers 3 to 7 in envelope will produce sounds that will go on until a 'PLAY 0,0,0,0' command is given in your program. The shape of the sounds produced by the envelope values is best shown in a diagram (Fig. 12.4).

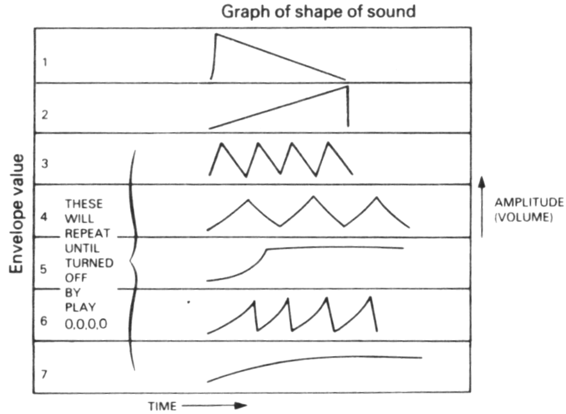


Fig. 12.4 Envelope values and note shapes

An envelope value of 1 gives a sound which starts abruptly, then dies away. A value of two gives the opposite: a sound which builds up in volume, then stops suddenly. Giving envelope a value of 3 makes a sound which *repeatedly* starts abruptly, then falls in volume, rather like 1 but at a faster rate. A value of 4 makes tones which rise in volume, then tail off, over and over again; 5 produces a sound which rises increasingly in volume to a level which is maintained until turned off, and 6 makes any sound produced by PLAY continue to rise in volume, then tail off abruptly, again repeating until turned off. Finally, 7 yields a note which rises more gradually to a peak than 5, but which is similar in that it will stay at the highest volume reached until turned off.

The program which follows will allow you to enter the envelope value you want in order to give you some idea of how the tones actually sound. All the sounds are made through channel 1 with an envelope period (see below) of 1000.

The tones are all note C (note value 1) in octave 3.

```
 10 CLS
 20 PRINT "ENTER ENVELOPE VALUE (1 to 7)";
 30 GET EV$: EV=VAL(EV$): IF EV<1 OR EV>7 THEN
    GOTO 10
 40 REM trap invalid values
 50 MUSIC 1,3,1,0: REM channel 1, octave 3, note
    C, wait for play
 60 PLAY 1,0,EV,1000: REM channel 1, no noise,
    envelope value, period
 70 WAIT 100
 80 PRINT "PRESS SPACE TO CONTINUE, OTHER KEY
    TO STOP";
 90 GET C$
100 IF C$=CHR$(32) THEN GOTO 10 ELSE
    PLAY 0,0,0,0
```

Envelope period

The envelope period allows you to change the relative length of time that tones take to reach their maximum peaks. So, for an envelope value of 2, a large value for the envelope period will make the note rise over a long period of time. Fig. 12.5 shows the effects of different envelopes.

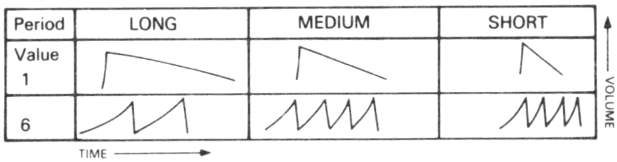


Fig. 12.5 Envelope period values on the envelope values 1 and 6

Varying the period of tones gives you control over the peak-to-peak frequency of the wave-forms produced by envelope values. In Fig. 12.5 you can see that a large value for the period means that a wave produced by an envelope of value 1 will produce a sound that starts abruptly, then dies away slowly. Reducing the envelope period makes the sound die away faster.

The program which follows will allow you to explore the effects of varying the envelope period.

After starting each tone, there is a WAIT whose length is calculated by taking the left-hand digit of the period and multiplying it by 100. This gives WAIT lengths of 100 for period of 1, 10, 100, 1000, etc. WAITs of 500 follow periods of 5, 50, 500, 5000, etc. You should try to alter these to suit your needs in mapping out this difficult terrain.

```
 1Ø FOR EV=1 TO 7: REM for all envelope
    values
 2Ø PRINT EV: REM display current value of
    envelope
 3Ø MUSIC 1,3,1,Ø: REM channel, octave 3, note
    C, volume Ø
 4Ø PLAY 1,Ø,EV,EP: REM play-channel 1, no
    noise, envelope, period
 5Ø A$=STR$(EP):L=LEN(A$): REM convert EP to
    a string
 6Ø A$=STR$(RIGHT$(A$,L-1)): REM fix for
    STR$() bug
 7Ø W=VAL(LEFT$(A$,1)): REM get left-hand
    number of EP
 8Ø WAIT W*1ØØ: REM wait - longer for long
    periods
 9Ø NEXT: REM next envelope value
1ØØ PLAY Ø,Ø,Ø,Ø: REM turn off sound
11Ø CLS
12Ø PRINT "EP=";: INPUT EP: REM get new sound
    envelope value
13Ø CLS: IF EP=Ø THEN END: REM enter zero to end
14Ø GOTO 1Ø: REM repeat above with new envelope
```

From paper to tune

This section shows you in detail how to get from sheet music to your Oric playing the tune. For the demonstration, part of a sea shanty has been used – it is up to the reader to identify the tune!

You have seen how each note has an associated letter and number and that the octave of a note has to be changed when you move up from B to C or down from C to B.

Note lengths are important in musical notation. They give the music its beat and rhythm. The shape of a note determines its length. The longest note is called a breve, but it is rarely used. Fig. 12.6 shows the note shapes and their relative lengths. You can see that each note is twice as long as its right-hand neighbour.

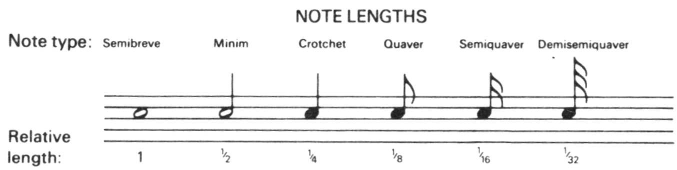


Fig. 12.6 Relative note lengths

Rather than use fractions for the note lengths, it is easier to give the notes arbitrary values. All that matters is that in the final PLAYing of the music, the relative durations of the notes should be preserved. That is, a minim should last twice as long as a crotchet and so on. You can always alter the WAIT lengths up or down after READing them in from DATA statements. Notes with a following dot (♩.) are $1\frac{1}{2}$ times normal length.

Fig. 12.7 shows the first eight bars of the piece. Although the music is in D minor, it has been translated for the Oric as if it were in C (no sharps or flats). Those with more musical backgrounds may be able to refine the rendition.

Fig. 12.7 also shows how to convert sheet music into Oric numbers. It shows the names of each note together with its Oric value, relative length and octave. Drawing up a table like this allows you to pin down any problems at an early stage and is clear and easy to follow, which reduces the likelihood of error.

Notice that most of the notes are in Octave 2, with the exception of notes 29, 30 and 31. Rather than use an octave datum for each note, which would make the DATA statements long and hard to follow, you can take advantage of the numerical positions of the notes and change the octave when twenty-eight notes have been played.

This could be done by an IF statement:

```
IF NN>=29 AND NN<=31 THEN OC=3 ELSE OC=2
```

However, IF can slow programs down quite a lot and the example uses a Boolean version of this which looks dreadful and is hard to unravel, but which works quickly:

```
OC=2-(NN>=29 AND NN<=31)
```

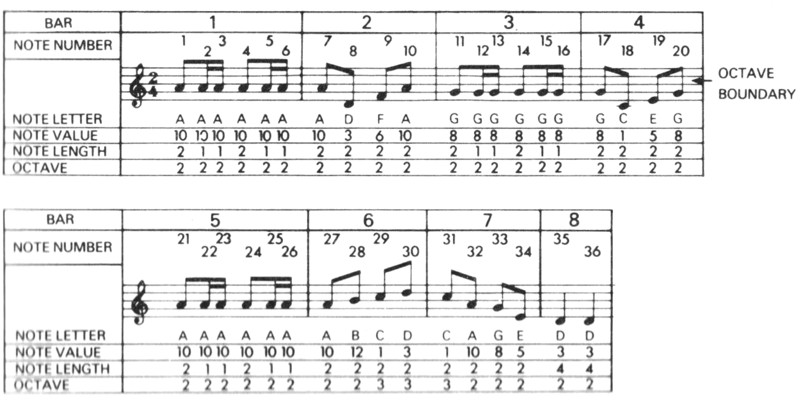
If the expression in brackets evaluates to TRUE (i.e. NN lies between 29

and 31) then C is defined as 2 + 1 = 3. Otherwise, if the expression evaluates as FALSE, OC is calculated as 2 − 0 = 2.

The sea shanty:

```
 1Ø REM music demonstration
 2Ø AN=36:OC=2 'NO NOTES;OCTAVE
 3Ø FOR EN=1 TO AN
 4Ø PLAY 1,Ø,Ø,Ø 'OPEN CHANNEL 1
 5Ø NN=NN+1 'COUNT NOTES
 55 OC=2-(NN>=29 AND NN<=31)
 6Ø READ N,L: REM get note and length
 7Ø MUSIC 1,OC,N,7'PLAY IT
 8Ø WAIT 5*(L*L) 'PAUSE
 85 PLAY Ø,Ø,Ø,Ø
 9Ø NEXT: REM loop till done
1ØØ REM ALL DONE
11Ø END
12Ø :'1ST BAR
13Ø DATA 1Ø,2,1Ø,1,1Ø,1,1Ø,2,1Ø,1,1Ø,1
14Ø :'2ND BAR
15Ø DATA 1Ø,2,3,2,6,2,1Ø,2
16Ø :'3RD BAR
17Ø DATA 8,2,8,1,8,1,8,2,8,1,8,1
18Ø :'4TH BAR
19Ø DATA 8,2,1,2,5,2,8,2
2ØØ :'5TH BAR
21Ø DATA 1Ø,2,1Ø,1,1Ø,1,1Ø,2,1Ø,1,1Ø,1
22Ø :'6TH BAR
23Ø DATA 1Ø,2,12,2,1,2,3,2
24Ø :'7TH BAR
25Ø DATA 1,2,1Ø,2,8,2,5,2
26Ø :'8TH BAR
27Ø DATA 3,4,3,4
```

The apostrophes in this program are short-forms for REM: see page 156.



BAR	1						2				3						4			
NOTE NUMBER	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
NOTE LETTER	A	A	A	A	A	A	A	D	F	A	G	G	G	G	G	G	G	C	E	G
NOTE VALUE	10	10	10	10	10	10	10	3	6	10	8	8	8	8	8	8	8	1	5	8
NOTE LENGTH	2	1	1	2	1	1	2	2	2	2	2	1	1	2	1	1	2	2	2	2
OCTAVE	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2

BAR	5						6				7				8	
NOTE NUMBER	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36
NOTE LETTER	A	A	A	A	A	A	A	B	C	D	C	A	G	E	D	D
NOTE VALUE	10	10	10	10	10	10	10	12	1	3	1	10	8	5	3	3
NOTE LENGTH	2	1	1	2	1	1	2	2	2	2	2	2	2	2	4	4
OCTAVE	2	2	2	2	2	2	2	2	3	3	3	2	2	2	2	2

Fig. 12.7 Part of a sea shanty

Miscellaneous

There are several interesting points about the Oric's sound-handling. For example, you are not restricted to the ranges given in the manual for PLAY. If you're not using SOUND or MUSIC, you can generate some useful sounds using PLAY on its own. The next example shows this point clearly. The routine picks random numbers for the PLAY function in the range 1 to 1000, displays them in the order given above, then PLAYs the result. After each sound, press the space bar for the next. This allows you to write down ones which you might want to use in your own programs. Press RETURN to end the program.

```
 1Ø HIMEM #17FF: REM #97FF for 48K owners.
 2Ø CLS
 3Ø DEF FNR(X)=INT(RND(1)*X)+1: REM random
    number function 1-X
 4Ø REPEAT
 5Ø TC=FNR(1ØØØ):NC=FNR(1ØØØ): REM tone and
    noise factors
 6Ø ES=FNR(1ØØØ):PD=FNR(1ØØØ): REM envelope and
    period factors
 7Ø PRINT TC;NC;ES;PD: REM display the values
 8Ø PLAY TC,NC,ES,PD: REM 'play' the values
 9Ø WAIT 1ØØ
1ØØ GET A$
11Ø UNTIL A$=CHR$(13): REM continue unless
    return pressed
12Ø PLAY Ø,Ø,Ø,Ø: REM turn off last sound
```

If you write a program which PLAYs music you will probably notice that the values given in the manual are not strictly true. If you enter values greater than the ranges given for either envelope value or envelope period, they will still have an effect. It seems that, as with the ASCII set, the envelope repeats itself in multiples of 7, so that 8 is the same as 1, 9 gives the same effect as 2 and so on. Again, the best way to find out exactly what your Oric will do is to experiment.

13. How to Debug Your Programs

Debugging Basic

This is one area of programming rarely covered in most programming texts. No one can tell you how to cure a given bug in general terms, but here are a few tips and ideas about delousing your programs.

Program bugs can be divided into three types:

1. Errors reported by the interpreter
2. Errors of program logic
3. Errors caused by quirks of the programming language

Sometimes, when you have a problem with a program, particularly when this concerns the flow of the program from line to line (as when you use numerous GOTOs and GOSUBs), it is useful to be able to follow exactly which line is being executed at any given point. The Oric has two words, which can only be used within a program, to let you do this. These words are TRON and TROFF (for TRace ON and TRace OFF).

When the Oric encounters the command TRON, it will display the line number of each line of a program as it is executed. When TROFF is encountered, this display is turned off. TRON is of most value when you cannot work out whether a particular line is ever being used.

The main problems with TRON are that it ruins any screen display you may have in your program and that it cannot be used to slow down the speed of execution of a program, so making all those numbers difficult to follow.

Interpreter-reported errors

These are the easiest to deal with since they have already been trapped – detected and located by the Oric. The Oric's Basic interpreter will report an error in a program with a question mark, an error message and a line number, as in '?SYNTAX ERROR IN 670'.

The first thing to do is LIST 670 (in this case) and to check its contents with examples in the Oric's Basic manual. The type of error usually provides a useful clue as to the cause of the problem. Syntax errors are most often caused by misspelling a word or missing out commas, semicolons or brackets.

If your problem is not a SYNTAX ERROR, then you'll have to start the detective work proper. There now follows a list of the Oric's error messages, their most common causes and cures (if any!).

ILLEGAL QUANTITY ERROR

This happens when you try to give a built-in function an illegal value. For example, SQR (which finds square roots for you) doesn't like negative numbers. This is not surprising since SQR(−1) is, in higher mathematics, an 'imaginary' number and it's hard to see how the Oric could use one of these. You'll get this error if you give ASC, SPC or CHR$ negative numbers or if you pass a number greater than 255 to CHR$. The easiest way to trap these sorts of errors is to make the value passed to these functions 'live' in a variable, which can be tested and set to some limit (upper or lower) in the case of an illegal value arising:

```
100 LET X= ...
110 IF X<0 OR X>255 THEN X=0
120 C$=CHR$(X)
```

This forces X to 0 if it's either less than 0 or greater than 255.

BAD SUBSCRIPT ERROR

This happens when you try to access a non-existent cell of an array:

```
10 DIM A$(15,9): REM set up array of 15
   rows by 9 columns
...
200 X=19
...
3000 LET V$=A$(X,7)
```

Line 3000 will cause this error, because the array only has fifteen rows and now you're asking for something from column 7 of the *nineteenth* row.

OUT OF DATA

You'll get this error when you've got more READ statements than elements of information in DATA statements. The most common causes of this are missing commas between items in DATA statements, missing items or too many READ statements. This is particularly difficult to track down if you're doing your READs in loops. The best thing to do is to put a PRINT after every READ so that you can see exactly where the problem is happening:

```
3000 FOR U=1 TO 30
3010 READ V$(U)
3020 PRINT V$(U): REM test line - remove when
     done
3030 NEXT
```

One way to avoid OUT OF DATA statements is to use a flag to indicate the last item and to READ the statements within a REPEAT ... UNTIL loop. Compare these two fragments of code:

A

```
10 REPEAT
20 READ A$
30 UNTIL A$="-999"
40 DATA A,B,C
50 DATA -999
```

B

```
10 ND=3
20 FOR I=1 TO ND
30 READ A$
40 NEXT
50 DATA A,B,C
```

A allows you to insert extra DATA statements between 40 and 50, and while B allows you to do the same, you would have to alter ND as well. A is not without its problems – it would be all too easy to miss out a DATA statement, have two commas next to each other, miss out a comma, etc.; all of these will cause problems later on, while the FOR . . . NEXT loop of B would at least trap some.

NEXT WITHOUT FOR and BAD UNTIL

These two problems are closely related. The first usually occurs when the order of nested loops has become confused. In the Oric's Basic, you don't have to give a variable to NEXT. Some Basics force you to do this:

```
1ØØØ FOR T=1 TO 4ØØ
1Ø1Ø NEXT T
```

This in fact makes an error more likely because otherwise the computer sorts out which variable is needed and so there is less chance of you making a mistake. However, it is useful to keep track of the order of variables in coming out of FOR . . . NEXT loops and you should always make a diagram of these by linking your program lines when working your routines out in rough. This allows you to spot the following where two loops are left in the wrong order:

```
FOR J=1 TO 1Ø

. . .

FOR K=1 TO 2Ø          ILLEGAL CROSSOVER

. . .

NEXT J

NEXT K
```

This will give you a NEXT WITHOUT FOR error message not because the number of NEXTs doesn't match the number of FOR . . . NEXT statements, but because you've got a 'NEXT J' which effectively cuts across the 'FOR K' inner loop.

A BAD UNTIL happens when your program has an UNTIL statement which is not matched by an earlier REPEAT command. One common source of this is too many GOTOs – do avoid these like the plague and if you must use them, make sure that they don't send control to some distant part of the program, or you'll have a devil of a job tracking down any problems you may get later. Always use a GOSUB in preference to a GOTO.

RETURN WITHOUT GOSUB

This is very similar to the two above. This error occurs when the interpreter meets a RETURN statement without having met a GOSUB – so obviously it doesn't know where to RETURN to. The most common cause of this error is forgetting to END a program, or redirect flow back to the start, so that the program 'falls off' your intended END into the subroutines.

You might do this, in a program whose outline looks like this:

```
10 REM start

...

100 GOSUB 1000
200 GOSUB 2000

...

1000 ...
1010 RETURN
2000 ...
2010 RETURN
```

There *should* be a line between 200 and 1000 which either ENDs the program or which directs flow back to line 10. Because there isn't, the program executes line 100 and GOSUBs to 1000, RETURNing to 200. It now GOSUBs to 2000 and RETURNs to the statement after the GOSUB in 200, which is shortly followed by a RETURN, without a matching GOSUB.

DIVISION BY ZERO

You cannot divide any number by zero. The reason for this is shown by dividing a number by increasingly small numbers:

10 ÷ 5 = 2	10 ÷ 0.5 = 20
10 ÷ 1 = 10	10 ÷ 0.00005 = 200,000

You can see that 10 divided by any number approaching zero would result in a very large number indeed and the Oric simply cannot handle this situation.

The way to avoid the error is to 'trap' possible zero values in divisions and force the result to zero or some other acceptable value. For example:

```
5000 IF DIV=0 THEN RESULT=0: GOTO 5020
5010 RESULT=NUMBER/DIV
5020 PRINT RESULT
```

This will at least ensure that the program won't crash due to a division-by-zero error. However, this is not strictly good practice, and you should put some reminder in the program somewhere to tell you or the user that this unforeseen circumstance has arisen.

OUT OF MEMORY

There are a number of possible reasons for this error message. It usually means that your program and its associated data require too much space and that if you want to continue to develop your program you'll have to make some savings somewhere. It can also occur if you're using more than sixteen FOR . . . NEXT loops inside one another (nested), or using too many subroutines in the sense that it would take more than sixteen RETURNs to come out of them all. It may also happen if you nest too many REPEAT . . . UNTIL loops.

In the former case you can make some space savings by removing REMs; spaces (e.g. condensing 'FOR T = 1 TO 100' to 'FORT=1TO100' will save 5 unnecessary bytes); re-using variables where possible (not a good idea, but you're probably grasping at straws!); making sure that arrays are DIMensioned no larger than they absolutely need be, and so on.

In fixing too many nested GOSUBs you'll have to change the structure of your program somewhat, perhaps using a GOTO instead of a GOSUB; a much more daunting task.

This error message can also occur if a GOTO puts program flow repeatedly through a REPEAT statement:

```
10 ...
100 REPEAT

...

200 GOTO 10
300 UNTIL ...
```

OVERFLOW

This happens when you try to use a number greater than '1.70141*10∧36'. It will therefore occur if the results of a calculation would exceed this number.

REDIMENSIONED ARRAY (REDIM'D ARRAY)

Once you have set up an array in Basic using the DIM statement, you cannot alter the size of the array. This means that you cannot use the same DIM command in a program more than once. It also means that you cannot reclaim the memory space used by a DIMension or array once its use is over. Always put your DIM statements in the first few lines of a program to avoid this.

STRING TOO LONG

Strings may not have more than 255 characters in them. This is most likely to happen when you add (concatenate) two longish strings together. You can prevent it happening by testing for the lengths of the strings to trap a total length which exceeds 255:

```
100 L=LEN(L$): R=LEN(R$)
110 IF L+R>255 THEN PRINT "TOO LONG": GOTO 130
120 A$=L$+R$
130 REM rest of program
```

An example of this is used in Chapter 7.

UNDEFINED STATEMENT

This will only occur if you try to GOTO or GOSUB to a non-existent line number in your program. The most common cause of this is when you've been tidying up your program, taking out REM statements, and a GOTO or GOSUB was pointing to one one of these. The cure is to replace the missing

line, perhaps with a dummy line, or to renumber so that the missing line is replaced.

UNDEFINED FUNCTION (UNDEF'D FUNCTION)

This error message will be displayed if a program tries to use a function which has not previously been defined by the DEF FN statement. Common causes are DEF FN statements skipped by GOTOs; wrong variables being used for function names; or a line having been erased by mistake.

REDO FROM START

This is an unusual error message since it will not 'crash' your program as the previous errors will. It happens when your program has instructed the computer to expect a number from the keyboard (e.g. INPUT AN), but the user has tried to enter a string (group of letters or some of the other symbols). The normal cure for this is to get all INPUT from the keyboard – not single key-presses (which are better collected by KEY$ or GET) as string INPUT – then convert the string to a number. This is achieved by use of the Basic word VAL. VAL converts a string to a number the number; will be zero if the string begins with a non-numeric character (except −, $, etc.), e.g.:

```
100 INPUT AN$: AN=VAL(AN$)
110 IF AN=0 THEN PRINT "NUMBERS ONLY PLEASE":
    GOTO 100
120 REM rest of program.
```

Of course, this will not allow the number zero! To get round this, you would have to check if the leading character of the string was zero. The ASCII code for zero is 48. This means that you could improve the above with:

```
105 IF ASC(LEFT$(AN$,1))=48 THEN GOTO 120
```

TYPE MISMATCH

String variables cannot be assigned numeric values. Nor can numeric variables be assigned to strings. This error message will be given to statements like 110 LET A$ = 3, or 120 LET A$ = Q, 7015 LET V = "WHO?" or 920 V = F$. LISTing the offending line will probably reveal that you've left out a string symbol ($) or even inserted an inappropriate one after a numeric variable name. This often occurs during editing a line using the cursor keys and inadvertently copying a $ symbol.

DISP TYPE MISMATCH

As you can see from its name, this message refers to some problem between the Oric's display mode (TEXT, LORES or HIRES) and a program's instructions. The Oric cannot use the command CIRCLE if a HIRES command is not in operation. In fact it will give this error message if your program uses any command that tries to access the HIRES screen while the Oric is in any of its text modes.

One possible cause for this error message is trying to use a HIRES instruction after a GRAB. You will need to issue a RELEASE to free up memory space for HIRES.

FORMULA TOO COMPLEX

One cause of this message is that you've somehow, possibly deliberately, used more than two IF ... THEN statements on a single program line. The only solution is to split the line into several sections over two or more lines. This will probably involve one IF ... THEN line setting a flag for another IF ... THEN on a later line.

Another possible cause for this error is that a mathematical expression uses too many levels of parentheses (brackets). In this case, the odds are that you would have a tough job deciphering it as well!

ILLEGAL DIRECT CALL

There are a number of the Oric's Basic words that can only be used within a program. INPUT and DATA are two of these. The usual reason for this message is when entering a program line having forgotten to use a line number.

CAN'T CONTINUE

You can break into one of your programs while it's running either by using CTRL C or by using the commands STOP and END as program commands. Any one of these will suspend the program and hand over control of the computer to the keyboard. You can then LIST parts of the program and have the computer display the values of variables, e.g. PRINT DY and so on. When you've tested the program's status in this way you can enter CONT (CONTinue) and the program will start up from where it was interrupted. If, however, you've made any alterations to the program during its suspension, by editing a line, then the Oric will give you this message. The reason it can't continue is that changing the contents of a program involve the Oric in altering its internal representation of your program, its variable and data storage.

Logical errors

These occur when your algorithms for solving a problem or performing some task are flawed. Check first exactly what you wanted to achieve, going back to your notes, flowcharts and systems designs. If you've just 'hacked' a program in at the keyboard and it doesn't work properly, the best thing to do is to pull the Oric's plug, make a cup of tea and sit down and DESIGN the program from scratch. Although it's great fun just sitting at the keyboard watching a program grow, quite literally at your fingertips, as you tweak a subroutine here and set a flag there, it's also the best recipe for a bug-ridden program whose flow and logic are so tangled that sorting them out will prove a nightmare.

One easy mistake to commit is to re-use variable names, confusing NR with RN, perhaps; using N and M as loop counters and as variables in their own right and so on. It may help to insert PRINT N statements throughout your program so that you can pinpoint where N is taking on inappropriate values. It helps too, if you're doing a number of these, to extend this to '"PRINT "N = "; N' or even 'PRINT "LINE 100, N = "; N'.

Logical errors are bound to be the hardest areas to give advice on as the way you tackle a problem will be markedly different from someone else. Several examples of logical errors follow as examples of the traps it's easy to fall into.

Relational operators

At the end of a game or educational program, the users are of course, told their score. They're also given some comment about their performance. In a test where the highest score (SC) is ten, can you spot the error in these lines:

```
1ØØØ IF SC<5 THEN PRINT "THAT'S NOT VERY GOOD"
1Ø1Ø IF SC>5 AND SC<9 THEN PRINT "THAT'S GOOD"
1Ø2Ø IF SC=1Ø THEN PRINT "FULL MARKS - WELL
     DONE"
```

If you can't spot the errors, they're at the end of this chapter.

Similar difficulties can arise over the use of the Boolean operators AND, OR and NOT. If you find yourself using several of these together, take extra care and draw up a 'truth table' to make sure that an expression performs the way you really want. The truth table for the statement

```
IF X=3 AND Y=7 THEN GOTO 5ØØØ
```

is:

X = 3?	Y = 7?	AND	GOTO 5000?
YES	YES		YES
YES	NO		NO
NO	YES		NO
NO	NO		NO

Truth table for IF (X = 3) AND (Y = 7) THEN GOTO 5000

The next diagram shows how the table alters when the AND is changed to OR

X = 3?	Y = 7?	OR	GOTO 5000?
YES	YES		YES
YES	NO		YES
NO	YES		YES
NO	NO		NO

Truth table for IF (X = 3) OR (Y = 7) THEN GOTO 5000

Setting out complex statements using Boolean operators like this should really be done during the design stage of a program, not used as an *ad hoc* debugging technique.

FOR . . . NEXT loops

You should make sure when using FOR . . . NEXT loops, that you don't jump out of them (using a GOTO) before the loop has reached its natural termination. This is one of the nastier features of Basic that no one tells you about. It's also one of the reasons for getting the hardest error to track; it can cause error messages to be produced in totally unrelated parts of a program. The example sorting program in Chapter 5 in fact commits this error, but since the early exit will be used rarely and there are only two loops involved, it's not a problem. If, however, you insist on jumping out of loops prematurely and use a large number of loops in a program, then beware spurious 'OUT OF MEMORY' and 'NEXT WITHOUT FOR' messages.

Jumping out of a FOR . . . NEXT loop looks like this:

```
190 FOR T=1 TO 100
200 IF QR(T)=IT THEN GOTO 220: REM leave if
    condition is true
210 NEXT
220 REM rest of program
```

A similar problem can arise (usually as a result of attempting to save space and time) by putting an entire loop on a single line:

```
210 FOR T=1 TO 100: IF QR(T)=IT THEN EF=
    1: NEXT
```

This line will 'work' in the sense that it won't give an error message when the interpreter obeys it. However, it's the same as the previous problem except that here the loop is 'fallen' out of when the IF test fails. The reason for this is that the NEXT is taken as part of the actions to be done ONLY when the IF test is true, i.e. if (in this case) QR(IT) = IT.

If you do need to jump out of FOR . . . NEXT loops prematurely, there are two ways of doing this 'safely': one is to set the counter or index to the upper limit, the other is to set it well above the upper limit, at the point at which you wish to quit. To use the examples given above, you could use either:

```
6000 FOR T=1 TO 10
6010 IF QR(T)=IT THEN T=10: REM condition met,
     set index T to limit (10)
6020 NEXT
```

or:

```
6000 FOR T=1 TO 10
6010 IF QR(T)=IT THEN T=99999: REM condition
     met, set index way above limit
6020 NEXT
```

Both of these 'fool' the Oric into thinking that the loop has finished, so it terminates it. This is a messy method, but avoids the problems that can be caused by using a GOTO to jump out past the NEXT.

Language quirks

This section deals with a fact of life in most Basics that isn't covered by most manuals. Put simply, it is that the body of a FOR . . . NEXT loop will *always* be executed at least once.

This probably won't bother you most of the time, since you'll be using well-defined limits for the counter, like 'FOR G = 1 TO 10'. However, when you get to using subroutines which will perform some action from a provided lower limit to an upper limit, for example:

```
9000 FOR IX=LL TO HL
9010 ...
9090 NEXT IX
9990 RETURN
```

and to which you pass the lower and upper limits (as in '100 LL = 10: HL = 890: GOSUB 9000'), you may find this to be a problem.

As a concrete example of this, let's consider using a subroutine to move the DATA pointer along a set of items in multiple DATA statements in order to collect a particular item. Normally you would do this by picking out specific items:

```
110 RESTORE: FOR DP=1 TO 9: READ A$: NEXT
120 READ A$
```

This moves the DATA pointer so that it's pointing at the tenth item (line 110), which is then READ into A$ in line 120. You don't want to have to do this for every DATA item you want, so you might use a subroutine that moves the DATA pointer to the item you want, then RETURNs so that you can READ the next item which is the one you need.

This might take the form:

```
9000 REM move data pointer to position N
9010 RESTORE: REM set pointer to start of data
     items
9020 FOR DP=1 TO N-1
9030 READ A$
9040 NEXT
9050 RETURN
```

This will work fine for all values of N except 1. This is because when N = 1, the FOR ... NEXT loop in 9020 becomes '9020 FOR DP = 1 TO 0'. The Oric interpreter doesn't realize that this is a nonsense, still READs one string from the list, advancing the DATA pointer one item, tests the upper limit with the lower in 9040, then RETURNs because the loop has finished (the index is greater than the upper limit). However, this means that you could *never* use this subroutine to access the first DATA item in a set of DATA statements.

Of course, there are ways round the problem; the following solutions will all work:

```
9010 RESTORE
9020 IF N=1 THEN RETURN
9030 FOR DP=1 TO N-1: READ A$: NEXT: RETURN
```

or

```
9000 RESTORE
9010 DP=1: IF N=DP THEN RETURN
9020 READ A$
9030 DP=DP+1
9040 IF DP<=N-1 THEN 9020
9050 RETURN
```

Answer to logical problem

What happens if SC = 5 or SC = 9?

14. How to Make Basic Programs Go Faster and Other Tricks

While Basic is not the fastest of languages, there are ways of making your Basic programs more efficient and hence slightly faster than is probably usual.

Oric speed

In the Oric, address 775 (#307) is used as a system variable that affects the cursor-repeat delay and the speed of operation of the Basic interpreter. POKEing 775 with a low number like 10 makes the cursor really skip across the screen – it reduces the delay time of the repeat. A low value like this also slows up the interpreter. You can prove this by RUNning a simple loop and timing it from a 'cold start' (switching on the Oric). Then POKE 775,10 and RUN the program again. It should RUN slightly slower. The normal content of 775 is decimal 139.

The other side of this coin is that POKEing address 775 with high numbers has the opposite effect. A value of 150 for example, slows the cursor-repeat drastically, but should show RUN time-savings of up to 20 per cent (depending on the operations in the program).

Putting these together, you could POKE 775,100 just before a calculation section to speed up its execution, then POKE 775,10 to speed up the cursor immediately prior to keyboard entry of data.

Physical structure

Curiously, most Basic interpreters waste a lot of time over GOSUB and GOTO calls. Most of them, and the Oric is no exception, put a RETURN address on a stack for GOSUBs (this is not necessary for GOTOs), then start looking for the destination line number from the *start* of the program. Naturally, this means that if you write your programs like most Basic programmers, with all the subroutines literally 'at the bottom', i.e. in the higher-numbered lines, then your program is spending a lot of time looking for those routines. Horrible as it may sound and feel to those brought up to this style, the most efficient place for your subroutines is in the *lowest*-numbered line numbers. This means that you could use the following approach to actually code your program (note the GOTO to 'skip' the variable definition section in line 90):

```
1Ø REM variable declarations and definitions
2Ø CX=48ØØØ: CY=Ø: BS$=CHR$(8): NL$=CHR$(13)
...
```

```
  9Ø GOTO 2ØØØ: REM jump to main program
 1ØØ REM subroutine section
 11Ø REM subroutine 1
 12Ø ...
 19Ø RETURN
 2ØØ REM subroutine 2
 21Ø ...
 29Ø RETURN
 ...
2ØØØ REM main program starts here
2Ø1Ø GOSUB 11Ø:REM subroutine 1
2Ø2Ø ...
```

Variables

Curiously, the Oric (as indeed, most Basic interpreters) can handle calculations faster if they're using variables, as opposed to numbers. This means that if you make much use of certain numbers, like the screen-start address, you can save a small amount of time by putting the values into variables and tossing these about. Such an approach does, however, act directly against any attempt at space-saving that you might want to make.

Integers

Programmers coming from other Basics may be used to using integer variables to gain some time. As mentioned earlier, this is not the case on the Oric. Use of integers actually slows the Oric down and they can't be used as loop indices (counters) in FOR . . . NEXT loops.

Subroutines

Subroutines can often be used to make savings in both space and time. For example, in Basic you can enter a subroutine at any point before the RETURN. This would allow you to define a subroutine to draw a border round the screen edges in say two parts: the first would draw the sides; the second would handle the top and bottom edges and a title. Then, whenever you only wanted to change the top and bottom borders, you would enter the subroutine at the second set of statements.

REMs

Since the Oric has to look at REM statements in order to ignore them they waste time. Therefore, once a program has been finished you can remove them all.

FOR . . . NEXT loops

FOR . . . NEXT loops can be accelerated slightly by leaving out the variable after the NEXT as in '100 NEXT' rather than '100 NEXT N'.

Variables again

The Oric makes up a table for itself in which to hold the addresses of variable values. This table operates on the FIFO (first in first out) principle, i.e. it takes longer to look up the value of a variable first used after another. This means that it's a good idea to 'declare' variables at the beginning of a program in order of frequency of use. Of course, you can't do this precisely, but you can approximate it. This would involve declaring variables that may only be used in part of a program at the start with a value of 0, just to get them an early entry in the table. Some languages force you to declare *all* variables at the start of a program.

Condensation

Together with knocking out REM statements (which saves both space and time) you should try to get as many statements on each line as possible. The Oric does limit you to eighty characters per line which may cause problems and you must be careful not to put a NEXT following an IF . . . THEN (unless it's planned . . .).

Keyboard interrupts

Finally, you can 'lock out' the keyboard by CALL #E6CA, which can give you up to 20 per cent faster execution of a program. If you do this (and it's useful for LPRINTing – see Chapter 15), you *must* remember to turn the keyboard back on by CALL #E804 before using GET or KEY$ and when the program has finished. You must *never* use a WAIT command with the keyboard disabled as your program would hang forever until you pressed RESET, which would put you back in direct mode.

Techniques for easier programming

Build yourself a library! Not literally, but on cassette or disk.

These would be general-purpose subroutines whose application is relevant to most programs. The library would include routines like PRINTing '"PRESS THE SPACE BAR TO CONTINUE"' at the middle of the bottom of the screen, waiting for a press of the space key to RETURN.

Other handy tools are things like sorting routines, and the advantage of these in a library is that many of them use substantial numbers of IF . . . THEN GOTO statements, which are the hardest statements to renumber manually.

One problem you may face when doing this is that the Oric has no 'merge' facility (version 1.1 does have this facility – see Chapter 2). This is used on other machines to add a program on tape to the end of one in memory. Most of them really do merge the programs, so that an incoming line will overwrite a line number in RAM of the same number. Knowing that the start and end addresses (for CSAVE) of a Basic program are held in #5F/#60 and #61/#62 should help you to write your own 'merger', using the Oric's facility for LOADing cassette files from a specified address. Do make sure that line numbers in the recorded program and in RAM are disjoint.

The simplest way to tackle this issue is by keeping all your subroutines at the

front of a program and LOADing the library before adding to it with a new program. This ties in well with the concepts outlined above for efficiency.

During program development at least, make full use of REM statements. They can always be deleted later – so make sure that you don't jump to them or you'll get UNDEFINED STATEMENT errors. One way to avoid this is to make a habit of starting all subroutines at round numbers; 100s or 1000s, according to needs, and to reserve the preceding line for the REM statement:

```
1999 REM sub for inputs
2ØØØ A$="":W$="":CP=POS(Ø)
```

The word REM serves as a REMark or REMinder to you the programmer and should help you make sense of an old program. REMs should also help other people understand what a part of your program is doing. It's a good idea to make full use of REMs particularly when you're starting to program.

The word REM can, in some circumstances, be replaced with the apostrophe. This happens when a REM is the last statement on a line; the two lines which follow are identical:

```
1Ø LET FJ=FJ+1:REM increment FJ
1Ø FJ=FJ+1: ' increment FJ
```

The first of the two lines takes up slightly more space in memory.

Unfortunately, the Oric is somewhat inconsistent in its use of the apostrophe as a replacement for REM; the following are illegal:

```
1ØØ FOR K=1 TO 1ØØ ' loop 1ØØ times
2ØØ NEXT: ' loop till done
```

Similarly, using an apostrophe as a REM after READ will also produce a SYNTAX ERROR.

A further problem with the apostrophe is that it cannot be used as the first 'statement' in a line. If you do put an apostrophe here, you will find that the Oric will 'lose' the line, so that it will not appear in your program. This is a nuisance, because it is fairly common practice to 'REM' out lines from time to time when debugging a program. Use of the apostrophe for this makes its removal very easy – but not on the Oric.

The answer to these problems is always to use REM, or to try out the apostrophe and to make a list of when it's all right to use it and when it's not. The memory savings hardly make this task worth while.

Using Boolean expressions

One of the most efficient time-saving techniques is to replace as many IF . . . THEN statements as possible with their Boolean equivalents.

For example, you might be using a variable Y whose value depends on another (say X), the relationship being expressed as:

```
1ØØØ IF X=3 THEN Y=12 ELSE Y=Ø
```

This can also be expressed as:

```
1ØØØ Y=-12*(X=3)
```

The latter takes much less time to be worked out when the program is running.

The Boolean expressions work like this: (X = 3) is evaluated as TRUE if X = 3. TRUE has a value of negative 1 (−1). Multiplying this by negative 12 (−12) gives 12, so when X = 3, Y will be set to 12. If X is not equal to 3, Y will be set to zero since when X = 3 is FALSE, the bracketed expression has the value zero. Multiplying this by any value will leave zero.

This technique allows you to reduce numerous IF . . . THEN statements to simpler forms, speeding up your program considerably. As a final example, when the relationships between two variables cannot be expressed simply and you have to use statements like:

```
1ØØØ IF X=3 THEN Y=12
1Ø1Ø IF X=9 THEN Y=2
1Ø2Ø IF X=1 THEN Y=-2
```

These can be condensed to:

```
1ØØØ Y=-12*(X=3)-2*(X=9)+2*(X=1)
```

The disadvantage of the technique is that such Boolean expressions are very difficult to unravel and hence debug, particularly if you've made a logical error in the statements.

Miscellaneous tips

PATTERN

Although only the lower six bits of any given screen byte can be displayed, PATTERN still gives an eight-bit effect. What it does is to approximate as closely as possible to a true eight-bit display by overlapping the value in PATTERN across bytes. This means that PATTERN 85 with its binary pattern of 01010101 might be spread across three bytes as shown in the diagram below. In the diagram, the pattern overlay begins at the fourth bit of the first screen byte and spreads across bytes 2 and 3. Notice that bits 6 and 7 are not used, but that bit 6 MUST be set. Were it not, and bit 5 were not set to 1, the byte would be taken as an attribute and not be displayed.

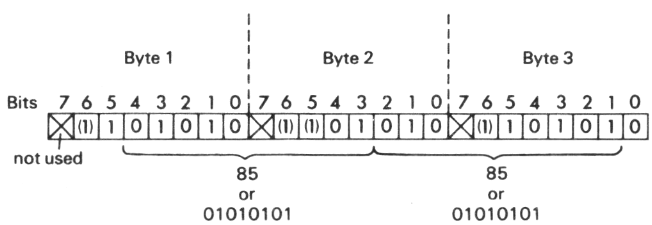


Fig. 14.1 How PATTERN 85 is mapped on to the six bits/byte of the Oric's HIRES screen display

The method works well most of the time, but certain column values coupled with other PATTERN values will cause the first few pixels of a line to be rather ragged. If this happens, you can get round the problem either by changing the PATTERN value or by altering the starting point of the line by a column or two, until you get the effect you want.

To find out more about PATTERN, RUN the following which draws a line according to a certain pattern, then shows the values in the screen bytes affected. You can then work out the binary patterns and explore this obscure area for yourself.

```
 10 HIRES
 20 FOR NP=1 TO 255
 30 PATTERN NP
 40 CURSET 0,0,0
 50 DRAW 20,0,1
 60 FOR A=0 TO 10
 70 PRINT PEEK(48040+A);
 80 NEXT
 90 GET A$
100 IF A$="S" THEN TEXT:LIST
110 HIRES
120 NEXT
```

Press the letter S to end the program.

Toggles and masking

As mentioned above in the SOUND section, address #26A (618 decimal) controls the status of a number of useful functions including keyclick on/off and cursor on/off. Each of the eight bits of the address has a specific function as shown in the table below:

Bit	*Function*	*Decimal*
0	Cursor on/off	1
1	Screen on/off	2
2	Printer on/off	4
3	Keyclick on/off	8
4	ESCape sequence	16
5	Protected columns	32
6	Double-height characters	64
7	?	128

These functions you may recognize as the first few control codes accessed by PRINT CHR$.

If the relevant bit of this address is set (to 1), then the function is 'on'. Unfortunately, exactly what constitutes on and off is not consistent across the functions.

To affect the bit which controls the function you want, you have to POKE 618 with a value which will not affect the other bits. This can be done by PEEKing the location, then applying one of the logical operators OR or AND NOT to the results.

For example, if you wanted to turn off the cursor, you would first work out which bit controlled it (bit 0), then the decimal equivalent of this number (1). Turning the cursor off would be accomplished by 'POKE 618,PEEK(618) AND NOT 1'. To turn it back on you would use 'POKE 618, PEEK(618) OR 1'.

Similarly, turning off the keyclick can be done by 'POKE 618,PEEK(618) OR 8', and turning it back on by 'POKE 618,PEEK(618) AND NOT 8'.

With some of the other bits, you may find that turning a function on needs the AND NOT operation, rather than the OR.

With a bit of judicious maths, you can affect one or more of the control bits at a time.

Notice the asymmetry of these two particular functions: to turn the cursor off you use AND NOT and to turn it on you use OR. The reverse is true for the keyclick: OR is used to enable the function, AND NOT to disenable it. You will have to experiment to find out exactly which logical operator applies to which function if you need to use this address and its associated functions to any great extent.

Location #20C holds some other useful bits: in particular functions such as CAPS LOCK, etc. are to be found here (bit 7). Similar techniques to those described above can be used to change these functions from within a program in order to ensure that the programmer retains complete control over the status of the machine at any given point in a program.

Restore

RESTORE is the command you use to set the DATA pointer back to the beginning of a set of DATA statements. Some machines offer a RESTORE LINE NUMBER facility, which allows you to set the DATA pointer to the beginning of *any* DATA statement. The Oric doesn't have this ability *per se*, but you can effect the same thing by DOKEing address 174 (decimal) with the relevant line number. The following program demonstrates the principle:

```
  1Ø PRINT "1=NAME/2=FRUIT/3=DRINK"
  2Ø A$=KEY$: IF A$="" THEN 1Ø
  3Ø A=ASC(A$): IF A<49 OR A>51 THEN 2Ø: REM
     49 is ASCII code for 1
  4Ø A=A-49: REM A is now Ø,1 or 2
  5Ø DOKE 174,1ØØØ+1Ø*A: REM restore line number
  6Ø FOR R=1 TO 3: READ A$: PRINT A$: NEXT
  7Ø CLS
  8Ø GOTO 1Ø
1ØØØ DATA ALBERT,BERTRAND,CHERYL
1Ø1Ø DATA APPLE,BANANA,CHERRY
1Ø2Ø DATA ALE,BEER,CIDER
```

Which key?

Normally, when you want to find out which key (if any) has been pressed, you would use either GET A$ (which waits for a key to be pressed, then puts the character of that key into A$) or KEY$ (which does not wait for a key-press, but otherwise behaves like GET). You may also use two addresses in RAM to get the information. The advantages of this are that you can test for key-presses much faster, especially if you want to check if a key is being held down. You are not restricted to the 'case' of the character either, since at least one of the locations always returns the same code, regardless of whether CAPS LOCK is on or off.

The following table shows the numbers you would get by PEEKing address 520 or DEEKing address 783.

Key	*PEEK (520)*	*DEEK (783)*	*Key*	*PEEK (520)*	*DEEK (783)*
1	168	47327	J	129	47614
2	178	47807	K	131	48126
3	184	47231	L	143	49149
4	154	47863	M	130	47870
5	144	47355	N	136	47357
6	138	47869	O	149	48635
7	128	47358	P	157	48631
8	135	49150	Q	177	47551
9	139	48125	R	145	47611
0	151	49147	S	182	48831
–	155	48199	T	137	47613
=	191	49023	U	133	48638
/	179	48063	V	152	47351
ESC	169	47583	W	190	48767
CTRL			X	176	47925
SHIFT			Y	134	48894
DEL	173	48607	Z	170	47839
RET	175	49119	(	189	48511
SPACE	132	48382	)	181	48575
A	174	48863	;	147	48123
B	146	47867	'	187	47999
C	186	47743	,	140	48381
D	185	47487	.	148	48379
E	158	48887	/	159	49143
F	153	47607	←	172	48351
G	150	48891	↓	180	48319
H	142	48893	↑	156	48375
I	141	48637	→	188	48255

There doesn't appear to be any rhyme or reason to the numbers in these particular locations. Suffice it to say that you can use PEEK (520) or DEEK (783) to find out which key is being pressed at any particular moment, without regard to the CAPS/lower-case mode, by testing the addresses given for the numbers in the table.

The reason for giving the two locations is that after holding a key down for a while, the contents of 783 change momentarily to a different number. The numbers in 520 don't alter like this, so that address is probably safer to use.

Program protection

If you want to make a completed program 'protect' itself from unwanted interference by others, i.e. prevent other people looking at your code by using CTRL C or RESET, then LISTing the program, you will need to take the following steps:

1. Make the first line of your program read:

```
1 DOKE #1B,#F42D: POKE #22B,#64
```

2. Save the program using the ,AUTO command, so that the program will RUN as soon as it has loaded.

The first line makes sure that if anyone presses CTRL C or the RESET button when the program is RUNning the Oric will either erase the program from its memory, or just hang up and not respond to the keyboard. Either way, no one (not even yourself) will be able to LIST the program. This means that you should only ever do this when you're quite sure that you've finished the program and will not want to change it again – because with these commands, you simply won't be able to!

Saving blocks of memory

The Oric can save blocks of memory to tape. This means that you can save TEXT screens of information, redefined character sets and HIRES pictures that you have written programs to draw. These could then be loaded by different programs. The advantage of this is that you don't need to have the picture-drawing routines etc. in the program that loads the information – thus saving space.

To save a block of memory to tape, you have to know the start and end addresses of the section that you wish to 'capture'. These are then passed to CSAVE command, using the letters A and E to signify start and end as in:

```
CSAVE "HIRESPICTURE",A#A000,E#BFE0
```

This would save a HIRES picture to tape. You can also add the ,S option to ensure that the data are more reliably recorded.

Unfortunately, the Oric has a problem with saving HIRES screens to tape. If you issue a CSAVE command when in HIRES, you may notice that a black line appears about two-thirds of the way down the HIRES screen. This is because the Oric is trying to display the message 'SAVING . . . *name*' on the TEXT screen. However, since you're in HIRES mode, the address at which it's dumping the message is still 48040, which is part-way through the HIRES screen. The message is overwriting the PAPER and INK values held in the first two cells of the HIRES row which starts at address 48040. This means that the PAPER and INK attributes are set to the default values of black and white respectively. There's really no simple cure for this bug. The best way to get round it would be to find out the values of the addresses in that row (48040 to 48079) using PEEK; store these in an array; put them in DATA statements in any program that's going to use the HIRES picture; then dump them back to the screen after the load, using POKE.

To save a TEXT screen, you would use:

```
CSAVE "TEXTSCRN",A#BB80, E#BFE0
```

Character sets may be saved using:

```
CSAVE "CHARSET",A#B400,E#B800
```

Because you can specify the start and end addresses of the chunk of memory to be saved, you could save portions of a HIRES screen, then merge them with other parts of different pictures.

To load blocks of RAM, use CLOAD as you did CSAVE:

```
CLOAD "TEXTSCRN",A#BB80, E#BFE0
```

15. Bugs in the Oric ROM

It seems as if the Oric was released before the ROM version of Basic (version 1.0) had had all its bugs sorted out. This means that you might sometimes write program lines that do not work as you think they ought to and you will not be able to find out why. The manual won't help because it was written on the basis that the ROM would function as it ought to and in several important respects it doesn't. Here's a list of the problems you may encounter and ways of dealing with them, commonly known as patches or fixes.

The TAB bug

TAB is one of the least standard of Basic words. On some machines, 'TAB(10)' can mean 'Move the cursor to column 10, unless it's already there or at a higher column number'. On other machines it might mean 'Move the cursor 10 TAB fields to the right'. In either case, TAB shouldn't overwrite anything already PRINTed on that row. TAB cannot move the cursor to the left.

Without modifications, TAB doesn't work very well on the Oric at all. It will always overwrite characters and doesn't do anything if you pass it a number less than 13.

The reason for the number 13 is that this is the number put into TAB's 'reference point'. If you must use TAB, execute a 'POKE #30, 0' before use, because #30 is where TAB looks for its base point. If you are going to use a lot of TABs, you might implement the following routine in your program which defines the exclamation mark to make TAB work like SPC:

```
10 DOKE #400,169: DOKE #402,#3085: POKE #404,96:
   DOKE #2F5,#400
```

Now, when you need a TAB, just put an exclamation mark first, as in

```
200 PRINT "SALARY:";:!: PRINT TAB (9);SA
```

A related problem is that if you are using an eighty-column printer, you will find that your Oric will only LPRINT sixty-seven characters per line (note that 67 = 80 − 13). To cure this, POKE #31, 93. (93 is 80 + 13). This should also allow you to tailor hard copy to a 40- or 132-column printer.

In fact, the best way to move the cursor on the current PRINT line is to POKE address #269 with the (decimal) column number in which you want the cursor to be, plus 2. For example, you might use;

```
100 PRINT "SALARY";: POKE #269,11: PRINT SA
```

This would display the word SALARY, move the cursor to the ninth column, then PRINT the value of the variable SA. The reason for adding 2 to the desired cursor column number is that the first two columns are reserved for the INK and PAPER attributes and must therefore be taken into account when calculating the relative positions of items on the PRINT line. Other ways of moving the cursor are given in Chapters 6 and 7.

This bug should be cured in version 1.1: TAB(9) should move the cursor to the ninth character position of a row. TAB cannot move the cursor backward: for this you would POKE #269 with the column required. Oric International do not make it clear how many of the system addresses like this one they have changed, so this dodge may not work on the Atmos or Orics with a version 1.1 ROM.

The IF . . . THEN . . . ELSE bug

This is a little documented but potentially highly frustrating problem. ELSE will work most of the time, but should you hit this one you would be hard put to it to work out what was going wrong. The problem is that the Oric puts the letter Y in front of the ELSE. This has the effect of occasionally altering the meaning of conditional statements.

Try the following:

```
10 A=1: AY=999: B=2
20 IF A<B THEN PRINT A ELSE PRINT B
```

Sure enough, your Oric will print the value 999, i.e. the value of the variable AY. What it does is to read line 20 as if it were:

```
20 IF A<B THEN PRINT AY ELSE PRINT B
```

This probably won't show up very often since most of your conditionals will do things like PRINT strings or direct program flow via GOTOs or GOSUBs. On version 1.1, this bug has been cured.

The patch for this bug is the following routine:

```
 10 DATA C9,C8,D0,05,20,61,CA,D0,07,C9
 20 DATA 27,F0,F7,4C,4E,EA,4C,E8,00,FF
 30 AD=#400
 40 REPEAT
 50 READ AV$
 60 AV=VAL("#"+AV$): REM convert Hex code to
    decimal
 70 POKE AD,AV
 80 AD=AD+1
 90 UNTIL AV=255
100 POKE #F0,1024
```

This will fix the problem but you will have to include it as part of any program which relies on the amended version of ELSE. Of course, there's nothing you can do about the fact that you can't use multiple statements with ELSE; e.g. the following is illegal (useful as it might be):

```
200 IF NB=0 THEN GOSUB 2000:GOSUB 3000 ELSE
    GOSUB 5000:GOSUB 6000
```

This is a great pity as it gives you great control over program flow.

The STR$ bug

STR$ is used when you want to convert a number to its string representation. This can be very useful for formatting numbers, getting their ASCII codes and so on. The problem with STR$ is that it adds a CTRL B, i.e. CHR$(2), to the front of all positive numbers. To get round this, you'll need a special routine after every STR$ usage like this:

```
1000 A$=STR$(A)
1010 IF A>=0 THEN A$=RIGHT$(A$,LEN(A$)-1)
```

or,

```
1000 A$=STR$(A): IF LEFT$(A$,1)=CHR$(2) THEN
     A$=RIGHT$(A$,LEN(A$)-1)
```

On version 1.1 this function still adds a leading character to a positive number converted into a string, but the character will now be a space (CHR$(32)), rather than CHR$(2) which used to make some numbers converted to strings appear in green INK. You may still have to strip off the leading characters of numbers passed to STR$ for some applications.

Most Basics add this space to the front of positive numbers, the reason being that they will then line up with negative numbers with a preceding – symbol.

Printout

If you're using a printer, you might have noticed that the Oric occasionally PRINTs a tilde symbol (~) instead of a character. What's happening here is that the keyboard-interrupt routine, which checks to see if a key has been pressed, sometimes sends the value #7F to the printer port. The best way round this is to turn off the interrupt sequence when printing, just as the Oric does when saving data to cassette. This will reduce the problem, but not eliminate it. To do this, simply insert a 'CALL #E6A' in your program. This turns off the scanning. Make sure that you don't then use a WAIT command because the Oric counts keyboard scans for this routine. If you use a WAIT now, you'll be waiting until you power off. To re-enable the keyboard, you must 'CALL# E804' which will start up the key-scan routine again. You can use the keyboard-disable routine at #E6CA to speed up your maths by up to 20 per cent because interrupts are generated at 20 cps (20 times a second). Don't forget to turn the keyboard back on however . . .

Turning off the keyboard before using a printer is not without its drawbacks. If you use 'CALL #E6CA' before LLIST, control will be returned (just as with LIST) to the keyboard – but the keyboard has been disabled. The only way round this is to use the RESET button!

Miscellaneous

The rest of this section covers problems that are not quite bugs, more gremlins. Version 1.1 of the Oric interpreter should perform better than version 1.0 in that some of the many system bugs have been cured.

GRAB and RELEASE

These are used to take or make available the HIRES page of memory for program and data storage. If you do some careful hunting, you'll find that the pointers set by these two routines vary from time to time, so that you actually lose a byte or two, i.e. the pointers are not set in exactly the same places. This is not a bug, rather just another symptom of a general malaise.

Loading problems

Sometimes, particularly if your volume is set too high, a program will appear to load correctly, but will not RUN. If you LIST the program and there are screenfuls of the letter U, then your Basic pointers have been corrupted. One way to relink them is to type in a dummy line, e.g. '1 REM', then immediately delete it. This should fix the program. If it doesn't, you'll have to reload with a lower volume setting. *Always* leave line 1 free so that you can use this dodge for faulty loads.

FILL

FILL is a useful command, particularly for setting attributes on the HIRES screen. Don't ever assume that FILL will actually move the cursor to the end of the last block filled. After a FILL command, use CURSET to put the cursor where you want it. This has been cured in version 1.1.

Numbers

Possibly related to STR$ (see above), the Oric prints numbers with a following space. This space has an ASCII code of 32. To avoid this, you'll have to convert numbers to strings, but watch out for the STR$ bug detailed above.

Escape sequences

Escape sequences to alter the INK or PAPER colours for subsequent text messages don't always work. For example;

```
PRINT CHR$(27)+"ETHIS SHOULD BE MAGENTA"
```

The message in quotes (not including the first character, which, after an 'escape' has been sent, is taken as an attribute control code) does not appear in the correct INK colour. However, if you insert a leading space it does:

```
PRINT CHR$(32)+CHR$(27)+"ETHIS IS MAGENTA"
```

Moral – don't use escape sequences, use PRINT CHR$ for all your control codes.

LIST

Sometimes, if you manage to enter LIST at the top of the screen, you'll either get a 'SYNTAX ERROR', or just OK – but without a LISTing of your program. Don't panic – your program's still there and the Oric hasn't crashed. Just type LIST again and *voilà*. There's no cure for this bug, it's just an annoying and occasional frustration.

HIMEM

There is a bug in the HIMEM function of the Oric. HIMEM should point well above the Basic program you're writing and well above the character set definitions, which are written into RAM when the Oric is turned on.

HIMEM is used to set the highest memory location that Basic programs and their data can use. It's provided to allow you to enter machine-code routines and to store them at the top of memory so that you can call them up from Basic programs. The idea of HIMEM is that your Basic programs will be prevented from writing into this area with memory-grabbing functions like DIM; so it protects an area of memory for use by machine-code routines.

Unfortunately, on power-up, HIMEM often points bang in the middle of the character set definitions. This has the unpleasant effect of distorting the characters should the space needed by your program and its associated variables get too large. When this happens, you'll suddenly find that some, if not all, of your PRINT statements are garbage. They will appear very odd indeed. You might even find that the contents of your arrays are being messed up if you are switching from HIRES to LORES. This is because when the Oric moves from TEXT to HIRES, it actually shifts the character set definitions to a new location. It does the reverse when going from HIRES to TEXT. Therefore, if HIMEM hasn't been set by the programmer, you're likely to overwrite the character definitions or to have some of your data overwritten. 16K owners should set HIMEM to #17FF.

This is done by using as one of the first few lines in a program the command:

```
10 HIMEM #17FF: REM set HIMEM out of harm's way
```

48K owners use #97FF.

FRE(0)

Another related bug is FRE(0). 'PRINT FRE(0)' should tell you how much spare memory there is for your Basic program etc. However, if you issue a HIMEM 1280, you'll be surprised to find that you apparently have some 65K to play with. Not so: 1280 is down in the Basic program start area and if you hope to make use of this magical extra memory you're mistaken. The extra memory is simply a figment of the Oric's imagination (try HIMEM 1284) and you really must set HIMEM to the values given above, unless you're using machine-code routines loaded in at some high address, in which case you'll have calculated the required value for HIMEM.

The PRINT separator

The screen of version 1.0 models can be thought of as divided into eight broad columns, starting at column positions 3, 9, 15 and so on. This was of little practical value, because the last band only has three columns. Version 1.1 uses five bands, each of eight character positions or columns. To see how the comma is used on your particular machine, try:

```
10 CLS
20 PRINT A,B,C,D,E
30 FOR I=1 TO 38: PRINT "'";:NEXT
```

This will give you some visual indication of the width of the columns.

Appendix A

A List of the Oric's Basic Keywords (in Alphabetical Order)

( ) indicates that the word must be followed by an expression in parentheses.

ABS( )
Returns the absolute value of an expression, i.e. disregards its sign: 'ABS(−4)' gives 4, 'ABS(4−10)' produces 6.

AND
A Boolean operator which is used to compare two expressions. Returns TRUE (−1) if, and only if, both expressions are true. AND is useful for the bit-wise masking of the contents of memory addresses.

ASC( )
Can only be used with strings, e.g. ASC(A$), ASC("G"). Returns the ASCII code of the string, or, in the case of longer strings, the ASCII code of the first character.

ATN( )
A trigonometric function which returns (in radians) the arctangent of the expression in brackets.

AUTO
A program-saving option. Can be added to CSAVE with a preceding comma to make a program RUN once loaded: e.g. 'CSAVE "CLARE", AUTO'.

CALL
Effectively a GOSUB to a machine-code routine at the address given. The Oric will execute instructions from that address until the machine-code equivalent of RETURN (#60) is encountered. CALL 62509 performs a cold boot (switches the machine on), CALL 555 performs a RESET. Try CALL 59150 (then CTRL T 3 times)!

CHAR
Used only in HIRES mode to place characters on the screen. Needs three arguments – the ASCII code of the character, 0 or 1 for the standard or alternate character sets and 0–3 for fore-/background colours.

CHR$( )

The complement of ASC( ). Returns a single character which is given by the number in brackets. CHR$(65) is the letter A.

CIRCLE

A HIRES command used to draw circles (well, ovals) on the HIRES screen, centred on the current cursor position. Needs two arguments: radius (1–119) and fore-/background (0–3). May not draw circles any part of which would not appear on the screen (except in one or two circumstances, due to a bug in the Oric ROM).

CLEAR

Nullifies all string variables, sets all numeric variables to zero.

CLOAD

Loads a program or block of memory into RAM. Examples: 'CLOAD' – load next program found on tape; 'CLOAD"" ' – load next program found on tape; 'CLOAD "BILL" ' – look for and load program called BILL; 'CLOAD "JILL",S' – look for and load a program called JILL, saved at 300 baud; 'CLOAD "CHARS",A46080,E47103,S' – look for and load a block of memory between the start (A) and end (E) addresses given, at 300 baud.

CLS

Used to clear the screen in any mode (TEXT, LORES or HIRES).

CONT

A command to be used in direct mode only. Used to continue a program after a break has been forced by STOP, CTRL C or END.

COS( )

Return the cosine of the argument given. Requires argument to be in radians, not degrees.

CSAVE

Used to send a block of RAM to a tape recorder. For saving programs or groups of bytes to tape. May be followed by various options: 'CSAVE "ZELDA" ' – save current program in RAM at 1200 baud, called ZELDA; 'CSAVE "RICHARD",S,AUTO' – save program in RAM at 300 baud, to RUN at once, next time it's loaded; 'CSAVE "TXT",A48000,E49119,S' – save current TEXT screen to tape under name TXT, at 300 baud.

CURMOV

A HIRES command to move the cursor relative to its present position. Requires three arguments: number of columns, number of rows and fore-/background colour (0–3). May be given negative numbers to move left or up.

CURSET

A HIRES command to move the cursor to a position specified by the first two arguments provided. Needs three arguments: column number (0–239), row number (0–200) and foreground/background (0–3).

DATA

Used to store information in a program to be accessed by the READ command. DATA statements may contain numeric or string expressions, as in 'DATA FRED,9,9,,2.3345,"JOHN",0'. If two commas are adjacent, a READ will result in a null value. String expressions do not have to be enclosed in quotes.

DEEK( )

A double-byte PEEK command; reveals the 16-bit contents of two adjacent addresses, the first being given by the value in brackets. DEEK(A) is the same as 'PEEK(A)+PEEK(A+1)*256'.

DEF

Must be followed either by FN or USR. DEF FN allows the user to define a function as in 'DEF FNA(R)=PI*R∧2'. Function names may be A to Z. Functions may only be defined with one expression (numeric) in brackets but may use and act on many variables. DEF USR is used as an alternative to CALL. 'DEF USR=1024' will pass control to a machine-code routine at #400 (reserved for such calls). USR is used when it is required that the routine should be passed a value and leave another in the accumulator.

The symbol ∧ means 'to the power of', so 'R∧2' is 'R squared'.

DIM

Sets up arrays in RAM. These may be string, numeric or integer; single, double or multi-dimensional. For example, 'DIM A(10),B$(10,5),A%(10,12,25)'.

DOKE

Double POKE. Used to place numbers larger than 255 into two-byte locations.

DRAW

A HIRES command to draw a line from wherever the cursor happens to be to a relative point specified by the following arguments. Requires three arguments: number of columns, number of rows and fore-/background (0–3). The first two arguments may be negative to DRAW lines to points to the left and/or upward from the cursor.

EDIT

Used to produce a program line for EDITing. LIST has a similar effect when given a single number. EDIT 100 is more or less the same as LIST 100.

ELSE

Follows an IF . . . THEN clause. Is itself followed by a series of instructions to be performed should the IF expression be evaluated as FALSE.

END
Stops a program and gives 'READY' message. Program may be resumed with CONT.

EXP()
Returns *e* to the power of the expression in brackets. Used in conjuction with LN (natural log) for logarithmic calculations.

EXPLODE
Fixed sound effect.

FALSE
Fixed constant with a value of 0.

FILL
A HIRES instruction needing three arguments. Fills an area of the screen defined by number of rows (1–200), by number of columns (1–239) with a value (0–127). Numbers must be integers. Most often used to fill an area of the screen with colour attributes (e.g. 'FILL 10,1,17').

FN
Invokes a function which must have been defined earlier using DEF FN: 'PRINT FNA(10)' would print the area of a circle of 10 units radius (assuming that A had been defined as in DEF above).

FOR
Marks the start of an iterative loop. Statements between FOR and the next NEXT will be repeated a specified number ot times. 'FOR Q = 1 TO 10' marks the start of a block of actions to be repeated 10 times. The index Q will be used to keep track of the number of times the loop has been executed. FOR can also be followed by STEP, which adds a constant to the index on each pass through the loop. For instance, for odd numbers, 'FOR Q = 1 TO 11 STEP 2'; for even numbers, 'FOR Q = 0 TO 10 STEP 2'. STEP may be a decimal fraction or, if the upper limit is lower than the lower limit, negative (to allow counting down), as in 'FOR W = 100 TO 1 STEP −0.5'.

FRE
May be used to find the number of bytes free for program and data storage as in 'PRINT FRE(0)'. May also be used to invoke 'garbage collection', i.e. to make the Oric reclaim storage area. (When variables are redefined they are not necessarily stored in the same locations as before. These unused locations, therefore, have to be reclaimed.) For instance, 'GC=FRE("")'. Don't use FRE("") in FOR . . . NEXT loops, as garbage collection can take time.

GET
Used to collect a single string character from the keyboard. Must be used with a string variable as in 'GET A$'. GET waits for a key-press.

GOSUB
Passes control to the line number specified, marking the point where program flow was altered. When a RETURN is encountered, program control returns to the statement after the GOSUB command. Can also be used with numeric variables as in 'GOSUB MENU'.

GOTO
Passes control to the line number specified, as in 'GOTO 1000'. May be given a numeric expression as in 'GOTO QUIT'. As with GOSUB, may be given expressions like 'GOTO T*10'. Following an IF . . . THEN clause, the GOTO may be left out as in 'IF Y = 10 THEN 500' which is the same as 'IF Y = 10 THEN GOTO 500'.

GRAB
GRAB makes a large portion of memory available to the user. This portion would be used were a HIRES instruction given, so the two are incompatible. That is, if you find that you need to GRAB space for a long program, you will not be able to use HIRES graphics. GRAB frees addresses #9800 to #B400 (48K) and #1800 to #3400 (16K) for program usage. RELEASE turns these bytes over to possible HIRES instructions.

HEX$( )
Will convert decimal expressions to hex format (base 16). For example, 'PRINT HEX$(27)' gives #1B.

HIMEM
Sets the highest RAM location for a Basic program and its data storage. Used to protect machine-code routines stored in the upper part of RAM. Should be set at the start of a program to #97FF (48K) or #17FF (16K) to avoid data or character corruption.

HIRES
Changes screen display parameters to 240 columns by 200 rows. Moves screen start address from 48000 to 40960 and moves start of character set definitions from 46080 to 38912 (48K), 6144 (16K).

IF
Start of conditional statement of form IF *expression* THEN *action*(*s*) as in 'IF X=3 THEN PRINT "THAT'S ALL, FOLKS": END'. Must be paired with a THEN statement.

INK
Used to set the foreground colour of either TEXT or HIRES screens. May be given value in range 0 to 7. Puts attribute in second column of screen, therefore will not affect items on screen whose attributes are defined locally.

INPUT
Used to collect more than one character from the keyboard. May be used to collect string (letters) or numeric values for variables e.g. 'INPUT A$ or 'INPUT N3'. Waits for a press of the RETURN key to mark end of data. May be used

with a PRINTed prompt as in 'INPUT "PLEASE ENTER YOUR NAME";N$'. Will give 'ERROR – REDO' message if expecting a numeric value (INPUT N) but user enters a letter.

KEY$
Used to collect a single character from the keyboard, as in 'K$=KEY$'. Does not wait for a key-press.

LEFT$()
Used to 'splice' strings or string variables. Needs two arguments: the string itself and the number of characters to be taken from the left-hand part of the string: e.g. 'P$=LEFT$("FRED",2)' will put 'FR' into P$. 'LF$=LEFT$(A$,3)' will put the leftmost three characters of A$ into LF$.

LEN()
FTXReturns the length of a string or string variable as in 'L=LEN(A$)' or 'L=LEN("RICHARD")'.

LET
An optional word which assigns values to numeric, string or integer variables (e.g. 'LET A$="PLEASE": LET P=10.5: LET Q%=3'). May be omitted.

LIST
Can be used in direct or program (deferred) mode. Will produce program lines on screen. May be used in a variety of ways: LIST will list the whole program (may be interrupted by pressing the space bar, repeat to continue). CTRL C will stop a listing. 'LIST −100' will list all lines up to and including line 100. 'LIST 200–300' will list lines 200 to 300 inclusive. LIST in a program will not return control to the program, but passes control to the keyboard when completed. 'LIST 1000 –' will list line 1000 to the end of the program.

LLIST
Sends program lines to printer. Used as with LIST. Does not check to see if printer is connected or turned on.

LN()
Returns the 'natural' logarithm of the expression in brackets, i.e. $\log_e x$. Can be used with EXP in logarithmic functions.

LOG()
Returns the logarithm to base 10 of the expression in brackets.

LORES
Used to select either of the low-resolution modes (black paper only). LORES 0 selects the normal character set (like TEXT), LORES 1 selects the alternate (teletext) character set. Use PLOT to display messages as scrolling the screen will cause reversion in the lower lines to TEXT mode.

LPRINT
Exactly the same as PRINT, but sends following items to printer, not screen. Can be used to send software switches such as condensed type to printer using control codes.

MID$( )
Used to get the middle characters of a string or string variable. MID$( ) needs three arguments: the string or string variable, the starting position and the number of characters to be taken (e.g. 'MID$("HELLO",3,3)' gives 'LLO'. 'MID$(F$,4,2)' gives two characters, the fourth and fifth, of F$).

MUSIC
An instruction to make the Oric produce a sound defined by the four arguments: channel (1–3), octave (0–6), note (1–12) and volume (0–15). No sound is produced if the last is 0; a subsequent PLAY may produce the sound required.

NEW
Clears the current program and all data from memory.

NEXT
Marks the end of a loop defined by FOR. May be followed by the relevant variable as in 'NEXT C', but this is optional and slower.

NOT
A Boolean operator which negates a following expression in a bit-wise manner. The decimal number 8 is represented as 00001000: NOT 8 would result in 11110111 (i.e. all the zeros change to 1 and vice versa). Used as an alternative to 'IF *variable <=expression* THEN . . .', as in 'IF NOT *variable > expression* THEN . . .' Most useful for masking various system flags.

ON
Used selectively to pass control to a series of line numbers (e.g. 'ON X GOTO 100,1000,2000,3000'. If X = 1, control will pass to line 100, if X=2, line 1000 will be executed and so on. If X=0, or is greater than 4, the statement will be ignored. Can also be followed by GOSUB. Very useful for condensing multiple IF . . . THEN GOTO statements.

OR
A Boolean operator which performs a bit-wise comparison between two expressions. Can be used with two numeric expressions as in 'IF X=3 OR Y=7 THEN GOTO 5000'. Here, if either X=3 or Y=7 then control will pass to 5000. Can also be used to mask system flags in much the same way as AND and NOT.

PAPER
Used in TEXT or HIRES mode to set the background colour. May be passed a number or variable with values in the range 0–7. Puts this attribute in the first column of the screen, not affecting items whose PAPER attribute is set locally.

PATTERN
Used in HIRES to set the format of lines produced using DRAW or CIRCLE. Maps an eight-bit value on to the six-bit display and allows the programmer to produce dashed or dotted lines etc. (e.g. 'PATTERN 231').

PEEK( )
Returns the contents of an address. Looks at a single byte of eight bits and therefore returns numbers between 0 and 255.

PI
A system constant used in geometric and other calculations. Is given as 3.14159625 (approximately 22/7).

PING
A predefined sound. May also be produced by PRINT CHR$(7).

PLAY
A complex command used to define sound production. Needs four arguments: tone channel(s) (0–7), noise channels(s) (0–7), envelope shape (1–7) and period (0–65535). 'PLAY 0,0,0,0' turns off all sound production.

PLOT
Used in either TEXT or LORES modes to place a message at a specified location on the screen. PLOT needs three arguments: the column number, the row number and a final numeric or string expression. For example, 'PLOT 10,12,"ARE YOU READY?"' will display the message starting from column 10 of row 12. If the last argument is numeric, then the ASCII character represented by that number will appear; e.g. 'PLOT 1,3,67' will produce the letter B at column 1 of row 3.

POINT( )
A HIRES function used to find out if a particular pixel on the HIRES screen is in foreground or background colour. POINT needs two arguments: the column and row numbers (in the ranges 0–239 and 0–199 respectively). POINT returns 0 for background and −1 (TRUE) for foreground colour.

POKE
Used to place a number between 0 and 255 in any of the Oric's RAM addresses, as in 'POKE 48000,65'. POKE doesn't work if the second value is preceded by a #. Use VAL to convert hex numbers to decimal first.

POP
Used to remove a RETURN address from the stack of such addresses. Used to return control to higher levels after nested GOSUB instructions.

POS( )
Returns the current column number of the cursor. The expression in brackets is a dummy argument; the instruction is usually used as 'POS(0)'.

PRINT
Used to display items on the TEXT or LORES screens. May be followed by a series of items to be displayed. These may be separated by commas or semi-colons, though the latter are optional. Examples: 'PRINT "GO NUMBER ";X', 'PRINT "IT'S YOUR"N"TIMES TABLE" ', 'PRINT "YOUR CURRENT BALANCE IS", BL'.

PULL
Rather like POP, PULL removes an UNTIL from its stack, terminating a REPEAT ... UNTIL loop before the condition in the UNTIL has been met.

READ
Use to extract information from DATA statements. READ must be given numeric variables for assignation of numeric items in DATA statements. Used as in 'READ A$,N,V$,N$(1,4)' etc. It doesn't matter where in a program the DATA statements are – they may come before or after any READ instructions.

RELEASE
The opposite to GRAB. Frees memory so that HIRES commands can be used.

REM
Used to remind programmers what obscure pieces of program are supposed to be doing. May, in some cases, be replaced by the apostrophe. All statements on the same line after a REM are ignored by the Oric.

REPEAT
Marks the beginning of an iterative loop whose termination will be when the condition following the next UNTIL instruction is evaluated to be TRUE. Used as an alternative to FOR ... NEXT when the precise number of passes through the loop cannot be predicted.

RESTORE
Sets the DATA pointer back to the first item in the first DATA statement.

RETURN
Marks the end of a subroutine invoked by GOSUB. Returns control to the statement after the GOSUB command.

RIGHT$( )
The complement to LEFT$( ). Takes the rightmost characters specified. Needs two arguments: the string itself (as a string or as a string variable) and the number of characters to take. Used as in 'A$=RIGHT$("LOGICAL",3)', which would put 'CAL' into A$.

RND( )
A system function that generates pseudo-random numbers. The argument in brackets will usually be 1 to give decimal fractions between 0 and 1. This is

then multiplied by an upper limit, and has 1 added to it to give numbers in the range 1 to N as in 'R=RND(1)*N+1'. May also be given 0 in the brackets to give the last number used as a random number, or a negative number in the brackets to reset the seed.

RUN
Used to begin execution of a Basic program. Sets all numeric variables to zero and nullifies all strings. May be used to jump into parts of a program as in 'RUN 2000', which will start program execution from line 2000.

SCRN()
The low resolution equivalent of POINT. Returns not just 0 or −1 but the contents of a TEXT or LORES screen position, given its column and row number. Needs two arguments, column number (0–38) and row (0–26). Used as in 'IF SCRN(10,12)=42 THEN PING', which will produce a sound if there is a star at column 10 of row 12.

SGN()
Returns one of three values: 1, 0 or −1. SGN tests an expression, and produces 1 if the expression evaluates to greater than 0, 0 if it evaluates to 0, and −1 should it come to less than 0. That is, 'SGN(3−4)' gives −1, 'SGN(4−4)' gives 0 and SGN(5−4) gives 1. Note that 'SGN (5>4)' will produce −1 because the expression evaluates as TRUE which has a value of −1.

SHOOT
A fixed sound.

SOUND
An instruction to produce a noise according to the three arguments: channel (0–7), pitch (0–65535) and volume (0–15). If the volume is set to 0, sound production will be delayed until an appropriate PLAY instruction is encountered.

SPC()
An instruction for the TEXT and LORES screens to PRINT the given number of spaces. Used always with PRINT as in 'PRINT SPC(10); "WELCOME" '.

SQR()
Returns the square root of the expression in brackets.

STEP
Used in FOR . . . NEXT loops to specify a constant to be added to the index on each traversal of the loop. (See FOR.)

STOP
Used to halt a program. Gives the message 'BREAK IN line number'. Control is passed to the keyboard and the program may be resumed (providing no editing has been performed) by CONT.

STR$
Used to convert numbers to their string representation. It also adds a leading character, the minus sign in the case of negative numbers and a CHR$(2) for others. This can cause problems, so needs removing before the string is displayed.

TAB( )
Used with PRINT, should move the cursor to the column number given in brackets. Probably the most imperfect function on the Oric. Add 13 to the column number required as a partial patch. TAB actually prints spaces, erasing items on a row if you're not careful.

TAN( )
Returns the tangent of the expression given in brackets. This should be expressed in radians, as should all trigonometric expressions evaluated by SIN, COS, TAN.

TEXT
Switches the screen into TEXT mode – 40 columns by 27 lines. Clears the screen and homes the cursor.

THEN
Part of an IF clause. Should the expression following an IF statement be found to be TRUE, then the instructions following the THEN statement will be executed, otherwise they will be ignored. (See IF and ELSE.)

TO
Used to set the upper limit of a FOR . . . NEXT loop; see FOR.

TROFF
Turns the trace function off. Only applicable within a program.

TRON
Turns the trace function on so that all line numbers are displayed as the statement(s) in them are executed. Only used as a deferred command, i.e. in a program.

TRUE
A system constant with the value −1.

UNTIL
Marks the terminating condition of a REPEAT loop. If the expression following the UNTIL is evaluated as true, then the loop will be terminated. Example: 'REPEAT: X=X+1: UNTIL X=10'.

USR
Used in two ways, one with DEF to define the entry address of a user defined machine-code subroutine as in 'DEF USR=1024'. The other use is to pass a variable to that routine as in 'USR(10)' which places 10 in the accumulator

for use by the routine. At the end of the routine, the accumulator holds some value produced as a result of the routine's actions.

`VAL()`
Used with strings or string variables only; returns the numeric value of any leading numbers. Handles negatives, hex and scientific notation. Returns 0 if the first character is non-numeric (with the exception of the – and # signs).

`WAIT`
Takes one argument, a numeric expression in the range 0–65535. Causes a pause in a program of N*10 milliseconds. 'WAIT 100' gives a pause of one second.

`ZAP`
Produces a predefined sound.

Appendix B

A Table of Useful Addresses

(Figures in brackets are chapter references)

Decimal	*Hex*	*Comments*
18 19	12 13	DOKE with 48000 + Row * Column. Used by PRINT routine to place items on TEXT screen. (6, 7, 8,)
27 28	1B 1C	Used by Interpreter as reference point. DOKE with #F42D to clear memory when CTRL C pressed. Usually #CBED. (14)
33 34 35	21 22 23	Instructions to jump to DEF USR routine.
49	31	Number of characters per line for LPRINT. Add 13. (15)
53	35	Cassette file name (if loaded) for next seventeen bytes.
95 96	5F 60	Start address for CSAVE.
97 98	61 62	End address for CSAVE.
154 155	9A 9B	Basic program start address.
156 157	9C 9D	Basic program end address.
162 163	A2 A3	HIMEM value.
174 175	AE AF	Current DATA statement line number. DOKE with line number to RESTORE to that DATA statement. (14)
520	208	Key pressed. 56 = No key. (14)
524	20C	Bit 7 controls CAPS LOCK. (14)
616	268	LORES/TEXT cursor row. (6,7,8)
617	269	LORES/TEXT cursor column. (6,7,8)
618	26A	Control codes. Keyclick/cursor, etc., on/off. (12,14)
619	26B	PAPER colour. (2, 6, 8, 10, 11)
620	26C	INK colour. (2, 6, 8, 10, 11)

Decimal	*Hex*	*Comments*	
621 622	26D 26E	TEXT screen origin.	(7)
623	26F	Number of rows/lines on TEXT/LORES screen.	(7)
630 631	276 277	Timer. Counts down at 100 counts/second. DOKE with 65535 for maximum.	(5)
775	307	Usually 39. 10 gives faster repeat on keys; 50 faster Basic.	(14)
783 784	30F 310	Key pressed.	(14)
38912	9800	Start of character set definitions (HIRES mode).	(12)
40960	A000	HIRES screen top left.	(10)
46080	B400	Start of character set (TEXT mode).	(9)
48000	BB80	LORES/TEXT screen top left.	(6, 7, 8)
49119	BFDF	End of HIRES/TEXT/LORES screen (bottom right).	(6, 7, 8)

Appendix C

Peripherals

You can attach a printer, such as a standard Epson or Oric's own MCP-40, via the third socket from the left of the back of the computer (looking from the front). This will allow you to get 'hard copy' of your program listings for example. LLIST can be used just like LIST to get program lines on paper. This is very useful for working on a program when away from the machine. You can also use a printer to get the results of calculations printed out and so on. To do this you use the command LPRINT exactly as you would use PRINT, the only difference being that the displayed messages appear on the printer, not the screen. You might also want to use the CTRL P toggle which should 'echo' all screen messages on the printer. If the Oric seems to hang up when you attach the printer, you've probably put the printer cable plug in the back of the computer upside down. The cable can be attached and detached while the computer is turned on without harm.

Oric International have been advertising disk drives for the Oric for a long time and these are now scheduled to appear in mid-1984. These are essential 'add-ons' for the enthusiast. They allow you to save programs on to a floppy disk rather than a tape. There are many many advantages to this. Programs can be saved and loaded in far less time and more reliably. The disk-operating system will tell you exactly what's on a disk so you can't lose programs on a long tape or forget where you put them. Disks occupy less space, even if you will probably need more of them than you would cassettes. One minor disadvantage with disks is that you are typically limited to 100K per disk, but with the more expensive systems this can rise to 200K. Twin disk drives are essential if you are to do any serious work on your Oric as you can transfer the contents of one disk to another in no time at all.

A company called ITL has produced a very good disk system for the Oric, called the Byte Drive 500. It remedies many of the Oric's problems and has some extremely sophisticated facilities normally found only on far more expensive packages. While the Oric disk-operating system seems to be little more than an extension of the cassette system, ITL's product offers many different types of file-handling and a wide range of programming tools.

As well as being able to see the Oric's display on a colour television, you could make use of its capacity to drive a colour monitor. A monitor is really a 'dedicated' VDU which cannot be used as a television. The socket for the RGB (Red, Green, Blue) output of the Oric is the second socket from the right at the back of the computer. Although monitors, particularly colour monitors, are quite expensive, they do give a picture of exceptional quality in terms of

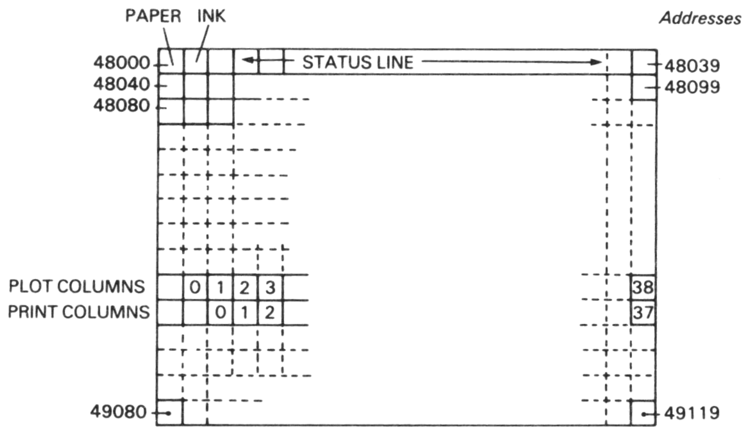
definition, colour and clarity. If you need to take photographs of your Oric's screen display, a colour monitor is essential.

You might soon be able to connect your Oric to the telephone lines via what is known as a modem (modulator/demodulator). This will allow you to communicate with other users and to participate in such systems as Prestel. The modem has been advertised for a long time now, but Oric International have yet to announce a release date.

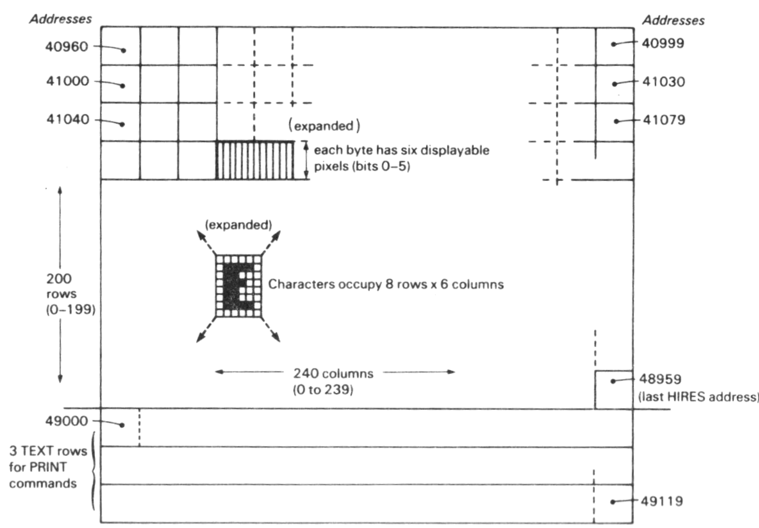
It would be surprising if no one were to begin manufacture of plug-in cartridges for the Oric. These would fit into the long multi-pin slot at the rear of the computer, next to the power socket. A cartridge contains a program which is immediately available once you turn the power on. The advantage of cartridges are many – they occupy no space in memory and 'load' instantly. Cartridges may contain games programs, but more usually this type of micro would have another language such as Forth, Lisp, micro Prolog, etc., or utilities such as a word-processing package, Basic toolkit/extension, assembler and so on.

Appendix D

TEXT/LORES screen map



HIRES screen map



INK and PAPER values

INK or PAPER	*Colour*	*Values to POKE* Foreground	Background
0	Black	0	16
1	Red	1	17
2	Green	2	18
3	Yellow	3	19
4	Blue	4	20
5	Magenta	5	21
6	Cyan (light blue)	6	22
7	White (buff)	7	23

Control codes and escape sequences

Press CTRL *and*	*Effect*	*PRINT CHR$*
A	Copy text to keyboard buffer	(1)
B	?	(2)
C	STOP program	(3)
D	Toggle PRINT items twice on/off	4
F	Toggle keyclick on/off	6
G	PING – 'Bell'	7
H	Move cursor 1 column left	8
I	Move cursor 1 column right	9
J	Move cursor down 1 row (line feed)	10
K	Move cursor up 1 line (reverse line feed)	11
L	Clear screen (form feed + 'home')	12
M	Carriage return + line feed	13
N	Clear row	14
O	Display off (until RETURN pressed)	(15)
P	Toggle printer echo on/off	16
Q	Toggle cursor on/off	17
S	Toggle VDU enable on/off	19
T	Toggle CAPS on/off	20
X	Clear line/keyboard buffer	(24)
Z	?	(26)
[	Start of escape sequence (same as ESC key)	27
]	Toggle protected columns on/off	29
Not available on keyboard	Move cursor to top left of screen (home)	30
		() = Not available

Most of these commands can be accessed both directly from the keyboard and by using a PRINT CHR$ statement in a program. Some only make sense when used directly at the keyboard (e.g. CTRL A for editing), while others are only accessible from PRINT CHR$ statements, e.g. 'PRINT CHR$(30)'.

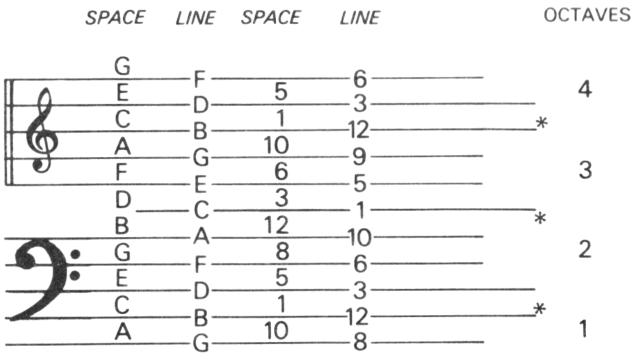
Table of ASCII character attribute codes for PRINT and PLOT

PLOT CHR$	*PRINT CHR$*	*Effect/Attribute*	*ESCape and*
0	128	Black ink	@
1	129	Red ink	A
2	130	Green ink	B
3	131	Yellow ink	C
4	132	Blue ink	D
5	133	Magenta ink	E
6	134	Cyan (light blue) ink	F
7	135	White (buff) ink	G
8	136	Single-height steady standard characters	H
9	137	Alternate character set	I
10	138	Double-height characters	J
11	139	Alternate character set	K
12	140	Flashing characters	L
13	141	Double-height alternate character set + flashing	M
14	142	Double-height + flashing normal characters	N
15	143	Double-height + alternate characters + flashing	O
16	144	Black paper	P
17	145	Red paper	Q
18	146	Green paper	R
19	147	Yellow paper	S
20	148	Blue paper	T
21	149	Magenta paper	U
22	150	Cyan paper	V
23	151	White paper	W
24	152	Text 60 Hz	X
25	153	Text 60 Hz	Y
26	154	Text 50 Hz	Z
27	155	Text 50 Hz	{
28	156	Graphics 60 Hz	|
29	157	Graphics 60 Hz	}
30	158	Graphics 50 Hz	~
31	159	Graphics 50 Hz	←

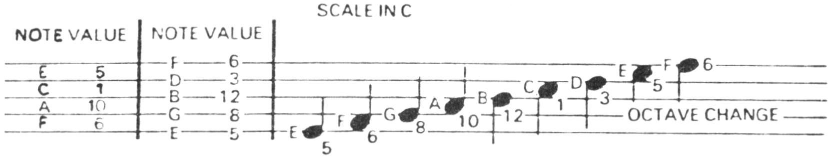
Appendix E

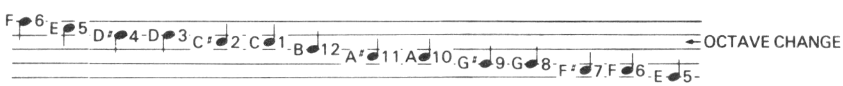
Tables of Music Values

NOTE	C	C♯	D	D♯	E	F	F♯	G	G♯	A	A♯	B
VALUE	1	2	3	4	5	6	7	8	9	10	11	12

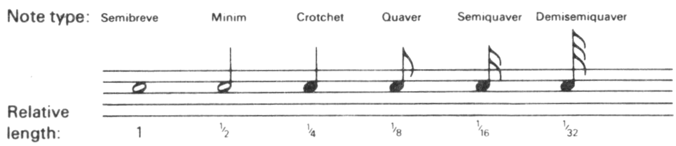


SCALE IN C





NOTE LENGTHS



Tone/Noise channel Value	Channels opened
0	none
1	1
2	2
3	1& 2
4	3
5	1&3
6	2&3
7	all

Appendix F

Oric Memory Maps

HIRES mode

48K Hex	48K Decimal	Description	16K Hex	16K Decimal
0	0	'Page 0'. See Appendix B for some useful addresses.	0	0
100	256	'Page 1'. System stack for data storage.	100	256
200	512	'Page 2'. System variables. See Appendix B for some useful addresses.	200	512
300	768	'Page 3'. Physical Input/Output (I/O) addresses.	300	768
400	1024	'Page 4'. Up to #420 for user's machine-code routines.	400	1024
500 97FF	1280 38911	User's programs and data storage space.	500 17FF	1280 6143
9800	38912	Standard character set.	1800	6144
9C00	39936	Alternate character set.	1C00 1FFF *'not available'*	7168 8191
A000	40960	HIRES screen.	A000	40960
BFE0	49120	Not used.	BFE0	49120
C000 FFFF	49152 65535	The Oric ROM.	C000 FFFF	49152 65535

TEXT mode

48K Hex	48K Decimal	Description	16K Hex	16K Decimal
0	0	'Page 0'. See Appendix B for some useful addresses.	0	0
100	256	'Page 1'. System stack for data storage.	100	256
200	512	'Page 2'. System variables. See Appendix B for some useful addresses.	200	512
300	768	'Page 3'. Physical Input/Output (1/0) addresses.	300	768
400	1024	'Page 4'. Up to ♯420 for user's machine-code routines.	400	1024
500	1280	User's programs and data storage space.	500	1280
			17FF	6143
B3FF	46079	*'not available'*		
B400	46080	Standard character set.	B400	46080
B800	47104	Alternate character set.	B800	47104
BB80	48000	Text screen.	BB80	48000
BFE0	49120	Not used.	BFE0	49120
C000	49152	The Oric ROM.	C000	49152
FFFF	65535		FFFF	65535

In most cases (apart from Pages 0 to 4 and ROM) conversion from 48K addresses to their 16K equivalents is given by subtracting 32768 (♯8000).

The text screen is apparently repeated in RAM – one screen lying between 48000 and 49119, another between 15232 and 16351 i.e. 48000 – ♯8000 and 49119 – ♯8000). You cannot, however, make use of both screens without corrupting a Basic program.

Appendix G

Summary of Version 1.1 Extensions to Version 1.0 of the Oric Basic

`CLOAD`

Used as with version 1.0, but two extra commands have been added: 'CLOAD "LIBRARY",J' – the ,J option will look for the program specified, then 'join' or merge the program with the program in memory. Line numbers in the two programs should be disjoint – i.e. if line numbers in the incoming program are the same as line numbers in the program in memory, there may be problems.

`CLOAD "THIS",V`

The ,V option verifies the program on tape against the program in memory. SAVE a program as usual, then rewind the tape and enter the command. The Oric will test the program on tape to make sure that it's the same as the program in memory. This can be combined with the ,S option to give you greater security.

`PRINT@`

Used to PRINT items at a specified point on the screen (TEXT and LORES modes only). Used in the form 'PRINT@ C,R,"MESSAGE"' (where C and R are the column (0–39) and row (0–26) values). Similar to PLOT, but allows items to be strung together in the command itself (unlike PLOT). Still won't allow inverse colours. Column 0 is now the very first screen column. PRINT@ is not, therefore, compatible with PLOT.

`RECALL`

Used to load arrays from tape (see STORE). Arrays must have been appropriately DIMensioned beforehand. String, real and integer arrays may be filled in this way. Used to save space in a program because you could write a short program, mainly of DATA statements, to fill an array, then save the array to tape. The data can then be loaded into another program which therefore doesn't need the DATA statements itself. Examples: 'RECALL A, "NUMBERS",S'; 'RECALL A$,"NAMES"';'RECALL A%, "INTEGERS"'.

`STORE`

Used to save the contents of string, real or integer arrays to tape (see RECALL). Examples: 'STORE A,"SALARIES"' – save the contents of array A to tape with the name SALARIES at 1200 baud; 'STORE A%,"PRIMES"'; 'STORE A$, "NAMES",S'. Typically would be used in a simple database program to save array contents after entering or altering data.

Note that, as with RECALL, you cannot use a string variable for the name of an array; you must specify the name in valid characters between speech marks. That is, 'RECALL A$, NM$' is invalid. This restricts use of the two facilities quite severely.

Version 1.1 of the Oric ROM displays information about the sort of file being loaded. The letter B means that a Basic program is being loaded, while C indicates a machine-code file.

When loading arrays, the letters S, I and R tell you whether the data are string, integer or real.

Appendix H

Table of ASCII codes

ASCII code	*character*	*ASCII code*	*character*	*ASCII code*	*character*
32	space	64	@	96	©
33	!	65	A	97	a
34	"	66	B	98	b
35	£	67	C	99	c
36	$	68	D	100	d
37	%	69	E	101	e
38	&	70	F	102	f
39	'	71	G	103	g
40	(	72	H	104	h
41	)	73	I	105	i
42	*	74	J	106	j
43	+	75	K	107	k
44	,	76	L	108	l
45	-	77	M	109	m
46	.	78	N	110	n
47	/	79	O	111	o
48	0	80	P	112	p
49	1	81	Q	113	q
50	2	82	R	114	r
51	3	83	S	115	s
52	4	84	T	116	t
53	5	85	U	117	u
54	6	86	V	118	v
55	7	87	W	119	w
56	8	88	X	120	x
57	9	89	Y	121	y
58	:	90	Z	122	z
59	;	91	[	123	{
60	<	92	\	124	¦
61	=	93	]	125	}
62	>	94	^	126	~
63	?	95	_	127	▒

SO YOU'VE JUST BOUGHT AN ORIC

But how does it work? What can it do? How can you use it for games? How do you write programs? How do you produce graphics? What do you do if something goes wrong?

Getting the Most from Your Oric assumes no prior knowledge on the part of the reader and can be used as a supplement to the Oric-1 manual. It will, however, take you much further in your understanding of the machine and programming in Basic.

There are chapters which explain and show you how to use the Oric's text and high-resolution graphics screens as well as chapters on sound generation and music. Each topic is accompanied by sample programs.

All aspects of programming are covered in comprehensive detail, from program design and development to debugging. Sample programs include games, music, graphics, utilities and corrections to some well-known Oric problems.

Oric-1 and Atmos owners will find the appendices (which include the RAM addresses of useful system variables, music translation tables, colour codes, ASCII character codes, memory maps, etc.), invaluable programming aids.

Cover design by Keith Pointing

U.K. £5.95
AUS
(rec
N.Z.
CAN

984 R17.95

Computing
ISBN 0 14
00.7808 8

GETTING THE MOST FROM YOUR **ORIC**

BRYAN SKINNER

ISBN 0 14
00.7808 8